Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Security of a NoSQL database: authentication, authorization and transport layer security

Bruno Jorge S. Rodrigues

Monograph presented as a partial requirement for
the conclusion of the Course of Computer Engineering

Supervisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Security of a NoSQL database: authentication, authorization and transport layer security

Bruno Jorge S. Rodrigues

Monograph presented as a partial requirement for
the conclusion of the Course of Computer Engineering

Prof. Dr. Rodrigo Bonifácio de Almeida (Supervisor)
CIC/UnB

Prof. Dr. Donald Knuth
Stanford University

Dr. Leslie Lamport
Microsoft Research

Prof. Dr. José Edil
Coordinator of the Course of Computer Engineering

Brasília, January 25, 2019

Dedication

This work is dedicated to all the people who supported me until here.

This is dedicated to God in the first place, for blessing me throughout this path that I have been crossing until here (1 Corinthians 10:31). This is dedicated to all my family, especially my father, Dimas, and my mother, Odilma, for loving me in first place, and for giving me strength, courage and the resources that I needed to get here without thinking twice. This is dedicated to Gabriella, for making me smile in my darkest moments, even when she was not in the best days as well. This is dedicated to all my friends, who made my days happier and also made this path easier to cross.

I love you all, and you are very important to me!

Acknowledgements

First of all, I thank God for His mercies and blessings, for His care and for each door that was opened and also for every door that was closed.

I also thank my father, my mother and the rest of my family, for every lesson taught, for helping me climb each step, for showing me love and for making me who I am today.

I thank Gabriella, for growing with me in many aspects, for hugging me when I most needed it and for always being there for me, and also all my closest friends, especially Fábio, Fernando and Ismael, for accompanying me during this path until here, and for helping me stand up when I stumbled.

I would also like to thank Prof. Dr. Rodrigo Bonifácio, for all the opportunities and support that he has been giving me, and also for the confidence that he has put in my work. I thank as well all the other teachers that helped me build all the knowledge that I have today, and, above all, who taught me to think.

Abstract

This project is intended to fill an important gap in a database management system called AngraDB: security. By using such systems, users need to be identified uniquely, so that their actions can be tracked and, most importantly, controlled. With that motivation, this work developed authentication and authorization schemes, following AngraDB's core idea of modularity and flexibility, and, given the sensitive nature of these systems, it has also implanted the SSL protocol upon the existing transport layer, all using the language Erlang, just as it is on the greatest part of the database project. These schemes were created considering other known databases as comparative models, such as MongoDB and CouchDB, and were also result of research about *de facto standards*, especially in regard to authentication and password hashing. Soon, this project will add some more security modules, such as data encryption and audit modules.

Keywords: security, authentication, authorization, access control, SSL, TLS, TCP, cryptography, digital certificate, public key infrastructure, PKI, password hashing, hash, database, DBMS, NoSQL

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

CA Certificate Authority.

CAs Certificate Authorities.

CRL Certificate Revocation List.

DAC Discretionary Access Control.

DBMS Database Management System.

MAC Message Authentication Code.

MD Message Digest.

MDC Manipulation Detection Code.

PBKDF2 Password-Based Key Derivation Function 2.

PKI Public Key Infrastructure.

RBAC Role-Based Access Control.

RIPEMD RACE Integrity Primitives Evaluation Message Digest.

SCRAM Salted Challenge Response Authentication Mechanism.

SHA Secure Hashing Algorithm.

SSL Secure Socket Layer.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

Chapter 1

Introduction

The amount of data that is processed daily over the world has already reached stratospheric levels, and it keeps growing, more and more. The conventional relational databases may not be the best option to deal with this data, since a great part of it is classified as non structured or semi structured data, such as emails, media files — audios, videos and images — and documents in general.

As an alternative to these relational databases, which are powered by SQL (Structured Query Language), emerged the concept of NoSQL, also known as "Not Only SQL". Databases that use this concept can either store structured or non structured data, and are optimized to handle data like this.

Many great companies invested in the development of NoSQL databases, like Google, Amazon and Apache. As a result of these efforts, we now have broadly used NoSQL databases, such as Amazon DynamoDB, Google BigTable, Apache CouchDB, Apache Cassandra, MongoDB and so the list goes.

Given this scenario, in 2016, a group a students from the Department of Computer Science of University of Brasília (CiC-UnB), under the leadership of Prof. Dr. Rodrigo Bonifácio, started the development of a project called AngraDB: a highly configurable, scalable, and highly available NoSQL database.

Among many semi categories of NoSQL, AngraDB fits one of the most known and adopted databases, which are the document-oriented ones: databases that focus on the storage of semi structured documents, the most common being JSON files. MongoDB and Apache CouchDB may be cited as very popular examples of document-oriented databases.

The project has been growing fastly. Besides the core functionalities, other plugins and modules were being developed, such as drivers to other languages like Java, Python and Ruby, an HTTP module, which turns possible to communicate with the core module via HTTP, and so on. Yet, this project holded a huge blank: security.

1.1 Problem Statement

Even though many people in this project have been putting effort on developing new functionalities for AngraDB, it still did not have any features related to security, in any ambit.

Authorization is one feature that is actually very common in many other database management systems (DBMSs). It relates to permission control, i.e., which actions a user can or cannot perform given some specific scope of the database. This relevant feature was still missing in our AngraDB project.

Unfortunately, the lack of an authorization mechanism comes together with other issues. In order to be able to give permissions for a specific user, or even check whether a user has or not some permission to execute a certain action, the system needs also to have the abstraction of user. In other words, our database also needed to be able to identify each user uniquely, so that we could manage authorization upon them. This particular issue can be included in a scope called **Authentication**.

Concluding this list of issues of this work, we have the problem that some user, when connected to AngraDB, might need to transmit some sensitive information. For example, when authenticating himself on the application, the user, depending on the authentication scheme, can send a password. This scenario makes it explicit the need of a **secure communication channel** between the user and the running database.

1.2 Objectives

The main objective of this work is to design and implement a set of security mechanisms on top of AngraDB. Given the nature of being highly configurable— based on the fact that AngraDB was conceived as a flexible and modular system—this work focused on the following items:

- Review the architecture of AngraDB core module, in order to analyze and state where to accommodate the changes that will be made.
- Prepare AngraDB to work with different implementations of both Authentication and Authorization modules, in order to maintain the characteristic of being highly configurable.
- Review and compare how other known database management systems (MongoDB and CouchDB) deal with the three security topics mentioned in the Problem section.
- Review some core concepts of cryptography that will be used both on authentication (for password hashing) and on establishing a secure communication channel.

- Review and bring the Secure Socket Layer (SSL)/Transport Layer Security (TLS) security protocol to AngraDB, and make it a possible configuration within our Database Management System (DBMS).
- Propose and implement one authorization module based on permissions and two different authentication modules, to reinforce the flexibility of AngraDB. One of which will be based on a challenge-response scheme, and the other one on digital certificates.

1.3 How This Document Is Organized

Besides the Introduction and Conclusion, this monograph is divided into the following chapters:

- *A Background About AngraDB* — this chapter will approach AngraDB, describing what it is, its story, its top-level architecture and how it communicates externally. Besides that, some Erlang concepts that are judged to be essential to this work are also approached;
- *Cryptography And Related* — this chapter will present some cryptography topics that were used as basis for some points of this work, using a more informal approach, to make it easier for someone who does not have previous knowledge in cryptography to understand, even if minimally, what was done in this project;
- *The Security Architecture of AngraDB* — this is the heart of this work. This chapter talks about AngraDB's security requirements and explains the top-level designs that were developed for each of these requirements. In the end of this chapter, a subjective evaluation of this security design is made through possible attack models;
- *The Implementation* — finally, details about the implementation of the whole security architecture and also of each mechanism that was developed are discussed in this chapter, including how the flexibility of the new security scheme was designed.

All the code that has been developed in this work can be found in the AngraDB repository, on the *security* branch (`https://github.com/Angra-DB/core/tree/security`).

Chapter 2

A Background About AngraDB

AngraDB is a Database Management System (DBMS) that features scalability, high availability and a flexible modular architecture, designed to be a greatly configurable system. It focuses on storing files in JSON format, and, hence, can be classified as a document-oriented database. Consequently, it fits in a broader concept of NoSQL databases, which are basically databases that can store either structured or non structured data.

Among some other projects inside AngraDB's repository (`https://github.com/Angra-DB/`), is the one called AngraDB Core. This, as the name sugest, is the main project of AngraDB.

The project Core is responsible for the creation and maintenance of processes that will handle and manage persistence itself, and processes that will handle connections that are made via Transmission Control Protocol (TCP) [5].

Because of AngraDB's goals in scalability and high availability, the language chosen for the construction of its core was Erlang [6], a functional programming language developed by Ericsson that makes it easy to create and manage processes, and also to handle concurrency.

2.1 AngraDB Core's Architecture

AngraDB Core's structure counts on an Erlang abstraction of supervisors and workers. Inside this concept, supervisors are processes whose main purpose is to watch their child processes, and take some predefined action if any of these processes die. Among the actions that might be taken in these occasions, the supervisor can simply instantiate another child, just as it can kill all the other child processes and instantiate new ones. On the other hand, workers are ordinary processes that can do whatever they were previously programmed to do.

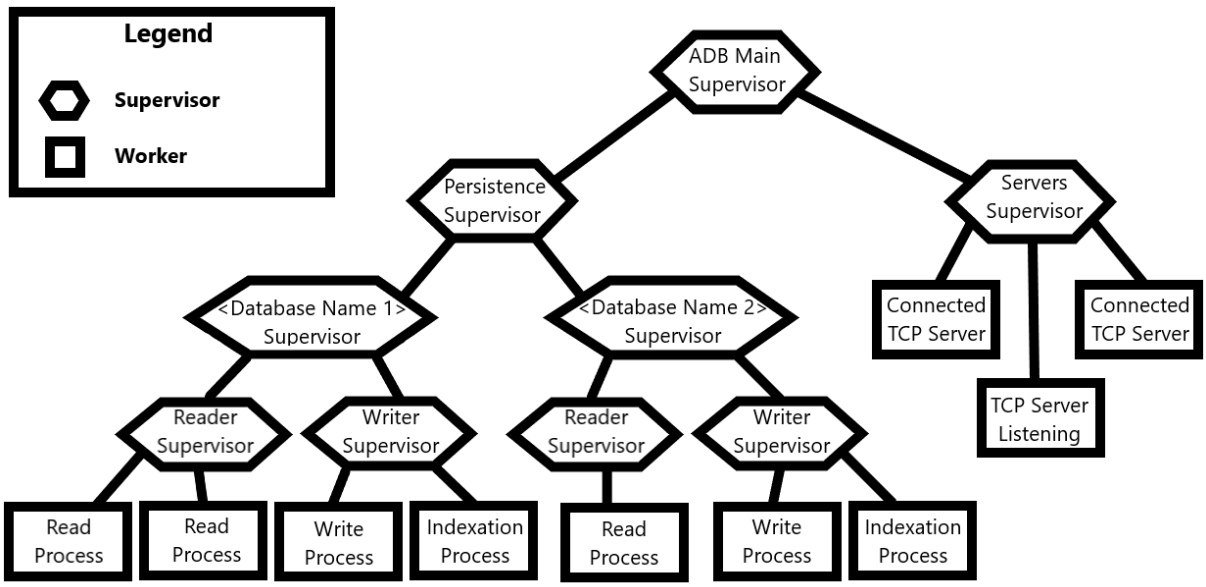

Figure 2.1: AngraDB Core's supervision tree.

Considering the previous explanation, Figure 2.1 and Table 2.1 of supervisors and their purposes, it is now possible to understand thoroughly how AngraDB Core is architectured.

The flow of process instantiation, as soon as AngraDB Core application starts, is to first initialize the main supervisor process, referred to in Figure 2.1 as "ADB Main Supervisor"; it immediately instantiates its two child processes: Persistence Supervisor and Server Supervisor. Soon thereafter, Server Supervisor starts the first socket server, which needs to be created right away so that it can listen to new connections as soon as possible.

The other processes shown in Figure 2.1 are created only when certain actions are taken at run-time. For example, a Database Supervisor is only started by Persistence Supervisor when a user asks to connect to a certain database. Right after created, Database Supervisor also instantiates Reader and Writer Supervisors. In turn, Reader and Write Supervisors, even though they seem to work in similar ways, they have an important difference: read operations can be performed concurrently, however, write operations cannot. Because of this, a Writer Supervisor instantiates only one process for write operations and another for indexation, while a Reader Supervisor instantiates a new read process whenever asked for a read action.

Table 2.1: Purpose of each supervisor of AngraDB Core.

Supervisor	Purpose
ADB Main Supervisor	Instantiate Persistence Supervisor and Servers Supervisor and watch them
Persistence Supervisor	Instantiate databases supervisors when asked to and then watch them
Server Supervisor	Instantiate a new TCP Server when a connection is established and watch these TCP servers
Database Supervisor	Instantiate a Reader Supervisor and a Writer Supervisor as soon as instantiated, and watch them
Read Supervisor	When asked to, instantiate a process that will deal with a read operation and watch them
Write Supervisors	Instantiate and watch two processes: one that will deal with write operations and another to deal with indexation

2.2 Important Erlang Concepts Used In AngraDB

Several concepts and resources from Erlang programming language helped the AngraDB team build the project. However, some of them were key to the development of this work, specifically.

The main concept that we are talking about is called "Behaviour". A Behaviour, to be a little more specific, is a design principle broadly used in applications that are developed using Erlang.

The core idea of a Behaviour is to reuse structures that need to be repeated many times along an application. By using the Behaviour concept, an Erlang common module — which is merely a source file written in Erlang and exports a handful of functions — is separated into two other modules, one of which is called "Behaviour Module" and, the other, "Callback Module".

A Behaviour Module, as the name suggests, is the module which will hold all the common behaviours of the structures of which there is a desire to reuse code. The specific parts of the source code will stay inside the Callback Modules. Inside a Callback Module, whose name is also suggestive, there will be defined the callback functions that are going to be used by the Behaviour Module. These callback functions are to be specified within the Behaviour Module itself. Thus, they can be interpreted as something like a contract between the Behaviour Module and the Callback Modules.

This Erlang "Behaviour" design principle looks a lot like the well-known Template Method design pattern [1], in which a base class defines a shared code skeleton, but lets some abstract (or concrete, but overridable) functions that are called inside this skeleton to be implemented by its sub-classes.

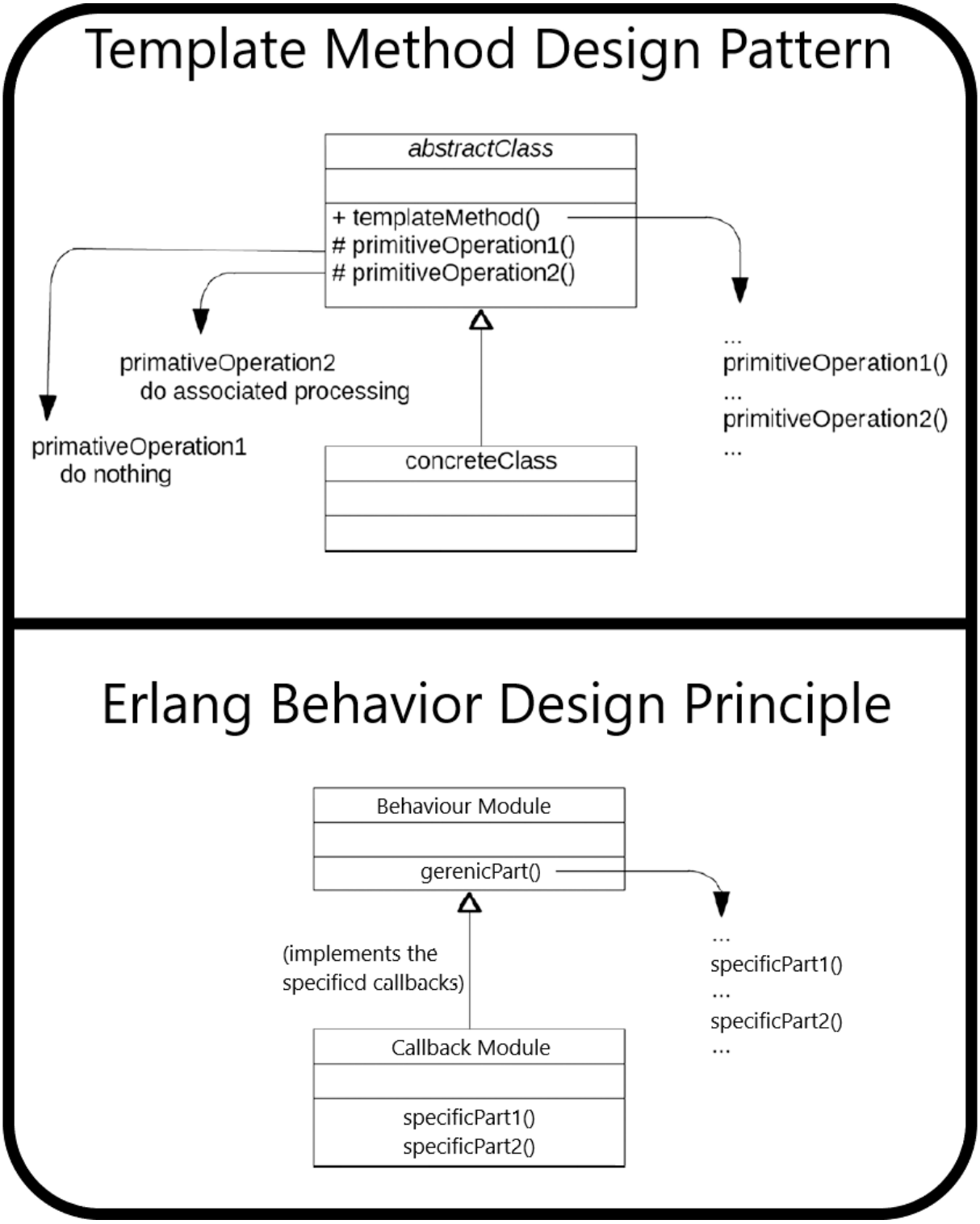

Figure 2.2: Comparation between Template Method and Erlang's Behaviour design patterns (Template Method UML taken from [1]).

Drawing a parallel to this line of thought, the behaviour modules would be like the base classes, by defining the skeleton of shared code, just as the callback modules would

be like the sub-classes, by implementing the specific functions that are to be called in the middle of the shared code. Figure 2.2 reinforces this comparation.

To bring this matter a little closer to our scenario, we can observe the recently mentioned Supervision Tree, illustrated in Figure 2.1, which founds the architecture of AngraDB Core. Inside this tree, many of these processes are supervisors, that is, their main purpose is to create and watch their child processes, and take a specific action when any of these processes die. Thus, the only basic differences between them are their child processes and those mentioned specific actions.

Having this said, it is simple to conclude that "Supervisor" should then be a behaviour module. "Supervisor" is actually a default behaviour module defined within Erlang/OTP. The job of the AngraDB Core team, when developing those supervisors, was to implement the callback functions that specify the child processes and the strategy used on the supervision of these very processes.

Going a little further on the usage of the "behaviour" concept inside AngraDB Core, we can also take all the worker processes from Figure 2.1 as an example of behaviour usage.

All these workers — Reader, Writer, Indexer and TCP Server modules — are actually servers (which might be obvious for the TCP Server module, but not for the rest). Each of these mentioned workers are callback modules of an Erlang default behaviour module called "gen_server", which stands for "Generic Server".

This Generic Server behaviour contains everything necessary to make it possible to trade messages with other generic servers or with any other application via sockets. The only must of the above mentioned modules is to tell what to do when a message is received, and they do that by implementing the behaviour module's specified callbacks.

2.3 How AngraDB Core Communicates With The World

We have just clarified the main internal architecture and some important Erlang concepts that were already in use in AngraDB Core and were also used during the development of this work, so, now, it is important for us to put in the table how the AngraDB Core establishes connections and communicates with the outside world.

Among some options of protocols to establish the client/server communication, such as User Datagram Protocol (UDP), the one that was chosen by the AngraDB Core team in the end was Transmission Control Protocol (TCP), which is a protocol from the transport layer of the TCP/IP network stack that takes care of handling the data exchange between hosts, and is the *de facto standard* for client/server communication over the network,

being widely used in other known applications, like MongoDB, Apache CouchDB, Oracle Database and so on.

Transmission Control Protocol (TCP) has some advantages over other protocols, like flow control, which prevents the hosts from receiving data faster than they can handle, guarantees the order and the integrity of the packets, and so on. All these features might be one of the reasons why this protocol is so widely used, since the other common option is UDP, which do not have any of these characteristics (but it is still an option because of the fact of being faster than TCP, which also comes from the lack of the mechanisms present in TCP). (More on TCP can be found in its specification [5])

The usage and implementation of this protocol, inside AngraDB Core, is thoroughly in charge of some of the previously mentioned workers that implement the "gen_server" behaviour module, as superficially explained in the last sections. Now it is time to check some details about these workers.

From the workers that implement callbacks for the gen_server behaviour, the ones that handle external communication via TCP are called "ADB Servers" inside our project.

As mentioned before, one process instance of a ADB Server is created right when the AngraDB Core application is initialized. This is done so that the core application can always listen to new TCP connections. Continuing the thought, this instance of ADB Server, when a new client asks to connect, begins the TCP handshake right away, and, if these handshake steps finish successfully, it immediately requests the Server Supervisor to instantiate a new ADB Server, so that this new process can listen to new connections while this previous ADB Server handles the communication with the client that just connected. If some error occurs during the handshake, that ADB Server dies and a new one is created, to keep on listening again. This process repeats whenever a new client tries to connect with AngraDB Core. A simple overview of this scheme can be seen on Figure 2.3.

Regarding to the implementation of these ADB Servers, the only things that needed to be done were setting the child specifications on the Server Supervisor, indicating what to instantiate when asked to start a child, and implementing the callbacks from the "gen_server" behaviour module that are used to handle incoming messages, in order to parse these very messages, and then interpret the commands inside them properly.

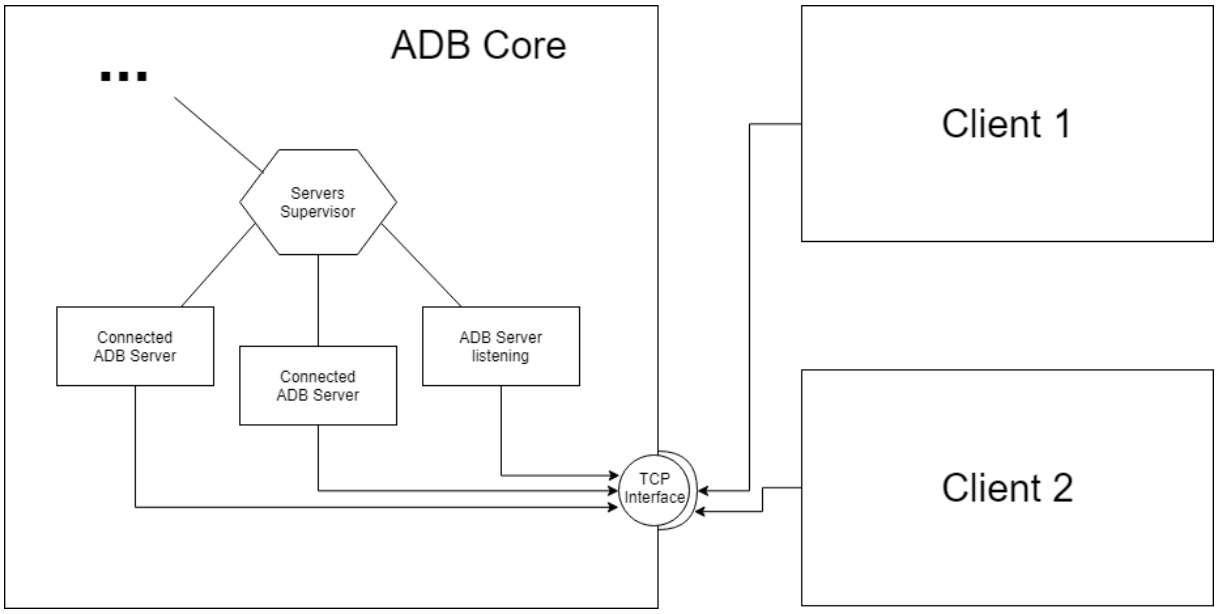

Figure 2.3: AngraDB Core's communication scheme.

Chapter 3

Cryptography And Related

Cryptography is considered very important to this work, since the pillars of almost every topic of this project — mainly authentication and transport layer security — are directly related to it. So, this chapter will be dedicated to cover objectively some Cryptography topics that were useful for this work.

These topics that were needed to build this project are: Cryptographic Hash Functions, which are necessary in the authentication scheme — for password hashing, to be more specific —, and Digital Certificates, which also involves Public-key Cryptography and Public Key Infrastructure (PKI), and they were all used to set up the transport layer security, and also used in the certificate-based authentication mechanism (more on that in the next chapter).

3.1 Introduction to Cryptography

The traditional concept of Cryptography emerged from the necessity of secrecy while transmitting messages over an insecure communication channel. In other words, the fundamental objective of Cryptography is to enable two (or more) parties to communicate over insecure media — which may be simply letters over the post service, or a telephone line, or a computer network — without letting untrusted people understand the content of the original messages that are being traded.

This is one of the most basic problems of cryptography. From it, a large number of encryption schemes — protocols that allow trusted parties to communicate secretly between them — were born. All these encryption schemes use to reproduce two steps: the encryption, which is a process applied by the sender party that turns the original messages (which are also called "plaintext") into cyphertext; and the decryption, which is, this time, applied by the receiver, transforming the cyphertext back into plaintext. Furthermore, to keep untrusted parties from figuring out the original content of the mes-

sages, both the encryption and decryption processes need an extra input: a key, which is an element that will not be known by eavesdroppers. This key can be the same for both the processes, originating a branch that is commonly called "symmetric cryptography", or can be unique for each of the processes, what can be called "asymmetric cryptography" or "public key cryptography". This is only one of the many classifications of encryption schemes, although it is also one of the most important ones. (more on [7], [8], [9], [10])

Even though cryptography arose as a way to trade messages stealthily in an insecure communication channel, many other subjects started to get encompassed to cryptography as well, such as digital signatures, certificates, message authenticity and integrity, pseudo-random generators, hashing, and so on.

3.2 Cryptographic Hash Functions

In order to understand the concept of Cryptographic Hash Functions, it is important to, first, understand what hashing is about, and also its applications.

A hash function itself has the fundamental objective of transforming or compressing its input — which is data in general, in different formats and different sizes — in a string called "digest" or "hash code", which will be outputted in a **fixed length**, acting as a fingerprint for this input data (as you can see in Figure 3.1). Decent hash functions tend to distribute the digests as uniformly as possible, and also to generate digests that look as random as possible.

Hash functions are very popular in many applications, algorithms and data structures, for example, in cases such as indexing in databases, which, in this case, has the objective of shortening the time needed to access information inside this database (see [11]); also, indexing in tables directly, in structures that are called Hash Tables, and so it goes.

Some other known uses of hash functions, such as ensuring the authenticity and integrity of messages , transactions and data in general (that is, hashing used for Message Authentication Code (MAC) and Manipulation Detection Code (MDC)), or "protecting" passwords inside a database — which is exactly the usage of hashing that we a re looking for in this work —, demand some special requirements, which are actually achieved by *cryptographic hash functions*.

Cryptographic hash functions, according to [12, p. 323-330] and [13], should satisfy three properties:

- *preimage-resistance* — for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x' such that $h(x') = y$ when given any y for which a corresponding input is not known.

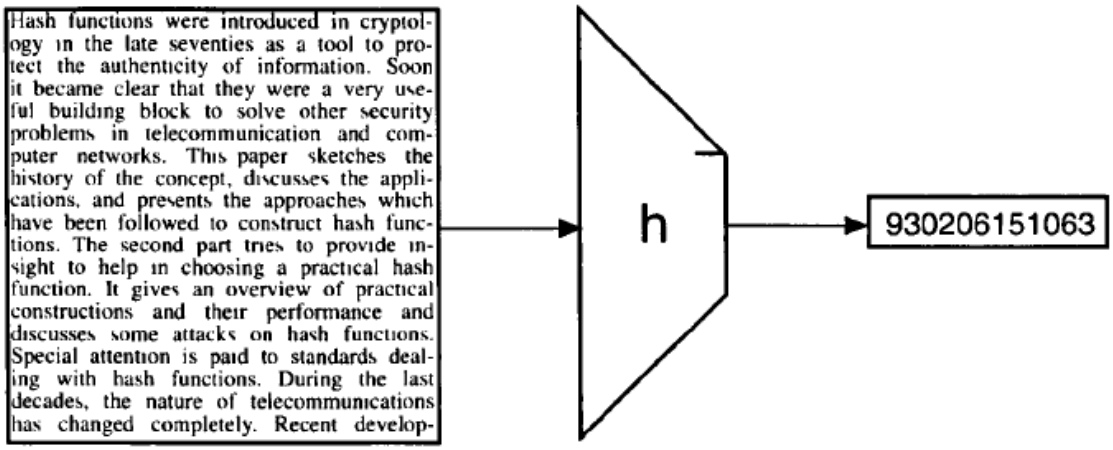

Figure 3.1: Example of a hash function (image taken from [2]).

- *2nd-preimage resistance* — it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a 2nd-preimage $x' \neq x$ such that $h(x) = h(x')$.
- *collision resistance* — it is computationally infeasible to find any two distinct inputs x, x' which hash to the same output, i.e., such that $h(x) = h(x')$.

Notes:

1. Collision resistance implies 2nd-preimage resistance of hash functions.
2. Collision resistance does not guarantee preimage resistance.

Alternate Terminology: Alternate terms used in the literature are as follows: preimage resistant ≡ one-way; 2nd-preimage resistance ≡ weak collision resistance; collision resistance ≡ strong collision resistance. [12].

Reminder: This work will take a more informal and objective approach. For formal definitions, examples of attacks on usages of cryptographic hash functions, types of forgery and so on, check [12], [13], [2].

Even though it is a point that ordinary hash functions may also achieve, the *randomness* is also a property that need to be present and reinforced in cryptographic hash functions. For example, see the Figure 3.2. It is possible to observe that, just because one letter changed to upper case, the whole digest changed. This point is important because it supports the *preimage-resistance* property, since it makes it harder — or even infeasible — to correlate information from the output to the input.

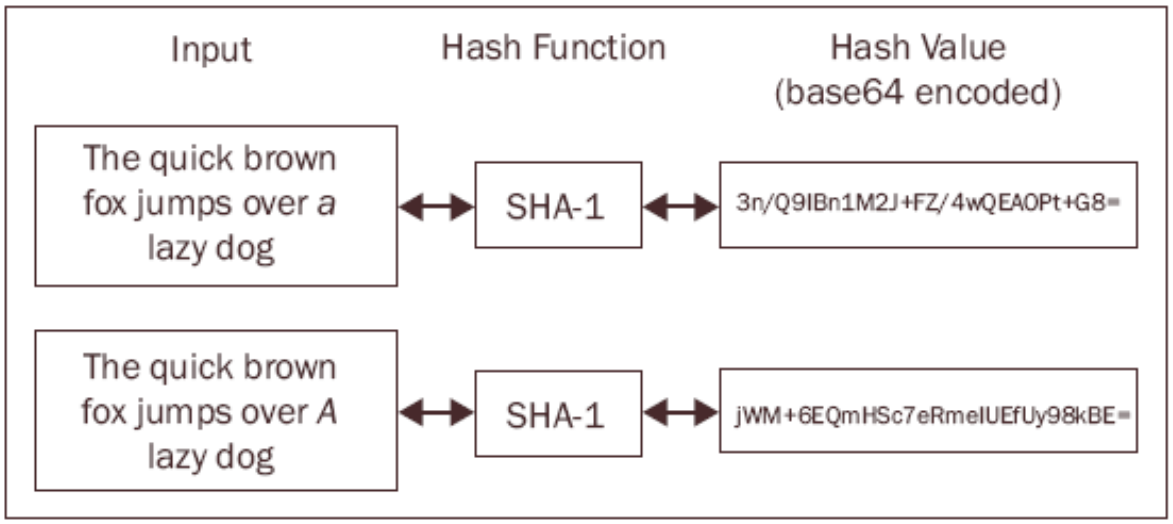

Figure 3.2: Simple example of the randomness of SHA-1, a widely known cryptographic hash function (image taken from [3]).

3.2.1 Cryptographic Hash Algorithms

There are many and many algorithms of cryptographic hash functions nowadays, but we can separate the most popular ones, which are: the Message Digest (MD) algorithm family (being MD4 and MD5 the most popular ones), which were developed by Ronald Rivest, and, of course, the Secure Hashing Algorithm (SHA) family (which got really popular with SHA-1, SHA-2 and SHA-3, the newest one), algorithms developed by National Institute of Standards and Technology (NIST) and supported by the USA Government — which is probably the reason of their popularity upon other algorithms such as the RIPEMDs, that were developed in parallel to the SHA family.

Notes:

1. For details on algorithms listed above and how they work, check [12, p. 344-351];
2. An interesting table with a longer list of cryptographic hash functions and comparisons between them can be found at [14].

Even being cited above as the most popular cryptographic hash functions, some of them have fallen into disuse, because of vulnerabilities and weaknesses that were being found along the years.

First, there was MD4, but many true weaknesses were being reported for this algorithm. Later, as an effort to try to strengthen MD4, MD5 was designed. In the beginning, it was well accepted, and became popular, but vulnerabilities came to the fore as well. A while later, collisions on both algorithms could be found in matter of minutes, even using

a regular computer. Once these collision attacks were found on MD4 and MD5, people tended to stop using them.

Based on MD5, National Institute of Standards and Technology (NIST) came up with the Secure Hashing Algorithm (SHA) (also called SHA-0), altering some points, such as the length of the digest (MD5 was 128 bits long, while Secure Hashing Algorithm (SHA) was 160 bits long), and, after few adjustments (only a little operation on its compression function), launched the well known SHA-1, which was the *de facto standard* for a relatively long period as well.

Since 2005, theoretical vulnerabilities on the SHA-1 were reported. Even though these vulnerabilities were only theoretical, it was enough for the community to start questioning the algorithm's security. Just for curiosity, only in 2017, was Google able to make a successful collision attack on SHA-1, by modifying a PDF without changing its SHA-1 digest, "killing" SHA-1 for good [15], [16], [17]. (Fore details on the weaknesses, vulnerabilities and collision attacks on MD5 and SHA-1, check [18]).

Later on the first theoretical vulnerabilities, big companies stopped supporting digital certificates with SHA-1, and started supporting its new and strengthened version, first launched in 2001, and was and is commonly called SHA-2. SHA-2 is actually a group of two different algorithms: SHA-256 and SHA-512, which differ on the sizes of the words, 32 bits and 64 bits respectively, and also on the length of the digests, which are 256 bits and 512 bits long, respectively. (There are also SHA-224 and SHA-384, which are respectively SHA-256 and SHA-512 with truncated digests, and SHA-512/224 and SHA-512/256, which are basically SHA-512 with digests truncated to 224 bits and 256 bits respectively)

Even though there were some theoretical reports of length extension attack ([19]) on a **weakened version** of SHA-2 (a version with less rounds performed), it is still the most used cryptographic hash function nowadays, and still there are no concrete evidences that could make it fall into disuse.

The newest member of the SHA family, SHA-3, does not have much to do with its older siblings, in terms of architecture of the algorithm itself. It emerged from a NIST hash function competition, where the Keccak algorithm came up as winner, and was then published as the SHA-3 Standard in 2015 (it is worth mentioning that SHA-3, as well as SHA-2, is able to generate digests in different lengths too, such as 224, 256, 384 and 512 bits). Currently, since it does not have any vulnerabilities reported yet, it is the most recommended hash function in terms of security, even though it still loses to SHA-2 in terms of popularity.

3.2.2 Password Hashing

Even sharing the same properties as cryptographic hash functions, algorithms used for password hashing still need some more special requirements. Basically, this subsection will tell what is needed in order to store passwords in a safer way, in case of Authentication mechanisms that indeed use passwords (which will be addressed in the next chapter).

As you can imagine, password hashing is the last security measure — or the last security layer — available. Some people even tend to classify it as a containment measure. Such things are said because, if some malicious party was able to overcome all the other security mechanisms, or even if there was a leakage and the database holding users' information became accessible by anyone, the only thing that will be between the passwords themselves and a malicious adversary is password hashing.

As said before, those three properties from cryptographic hash functions are still needed in password hashing, but, besides that, when hashing a password, there are some other features that need to be present and steps that should be followed, in order to be immune to certain attacks ([20]):

- An algorithm for password hashing **needs to be slow**, that is, computationally expensive. It might look a little confusing at first, even because ordinary hash functions in theory should to be easy to calculate, but in this case it is a little different. Computers nowadays have processing power to calculate billions of hashes in a minute. Therefore, in order to compromise the feasibility of attacks such as the *Brute Force* (which is an attack where many combinations are tried as inputs of the hash function in use, until the output matches with the desired hash value), this property needs to be satisfied.

- It is important to append a random string to the password before hashing and storing it in the database. This random string, which should be generated **per password**, is commonly called *"Salt"*, and it is often stored in plain text along with the hashed password (which had the salt appended before being hashed). This step prevents some attacks, such as the *Rainbow Table* (an attack where the adversary has a table with pre-computed hash values of many common passwords, to see if some of his hash values match with any other inside the leaked or invaded passwords database).

Note: More about these above mentioned and other attacks can be checked in the section 4.4 (Evaluation Based On Attack Models).

The technique used to slow down existing hash algorithms is called *key-stretching*. Some popular examples of *key-stretching* algorithms are Password-Based Key Derivation

Function 2 (PBKDF2), scrypt and bcrypt, being PBKDF2 the most widely used. Technically, bcrypt is not only a key-stretching algorithm itself, it is indeed a hash function made for password hashing, with parameters to make the computing more expansive (by setting an iteration count) and to receive a *salt*, but the community use to reference these three algorithms together. Other than bcrypt, PBKDF2 and scrypt are purely *key-stretching* algorithms; these functions basically receive, as parameters, the hash function (which may be one the mentioned above, like SHA-512, for instance), the input data for the hash function — that is, the password —, the *salt* that will be appended to the password before the operation, and, last but not least, the iteration count, which basically tells how many times this function will repeat the hash execution (of course, the larger this iteration count is, the slower will be the calculation of the final hash value).

Having all this said, it is possible to conclude that, for password hashing, which will be the case when implementing Authentication in AngraDB Core, a good combination would be SHA-512 or SHA-3 along with PBKDF2 or scrypt, having a different salt per password and also good values set for the iteration count (some value that can really slow down the calculation of the final hash value).

3.3 Public Key Cryptography, Digital Certificates and Public Key Infrastructure

Digital certificates are essential for some matters, such as Transport Layer Security or certificate-based authentication, which are two important topics of this work. However, to first understand digital certificates, it is relevant to have a background on Public-Key Cryptography, since it is basically what powers digital certificates, and also indicates what these certificates are for. Furthermore, Public Key Infrastructure (PKI) will be presented, in order to explain and reinforce the reliability of these certificates.

3.3.1 Public Key Cryptography

As introduced in the beginning of this chapter, public key cryptography — or asymmetric-key cryptography — is basically a system in which the key that is used to encrypt information (or validate signatures), the public key, is different from the key that will be used for the decryption (or to digitally sign), the private key (see Figure 3.3). Even though these keys are different from each other, they are mathematically related, and one key cannot be forged using the other, i.e., it is infeasible to forge the private key using the public key. ([7], [21], [12])

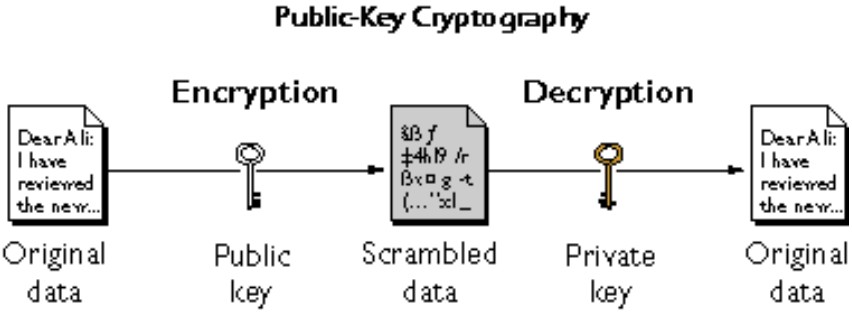

Figure 3.3: Illustration of a public key encryption scheme (image taken from IBM Knowledge Base).

The public key, as the name suggests, is publicly available — it can be held in a public repository, for example —, while the private key is only accessed by its owner. This fact is important because it simplifies a lot the key exchange process, which is needed in symmetric-key cryptography.

This feature makes public key cryptography suitable for confidentiality or privacy. However, public key cryptography alone does not work for authentication, because, since one of the keys is public, anyone can make use of it, indistinguishably.

Talking about its disadvantages, the one that stands out the most is the fact that public key cryptography algorithms use to be much slower when compared to the symmetric-key ones. Given this point, when the objective is to encrypt bulky data, public key cryptography is used just for the exchange of the symmetric keys, which then take care of the encryption of the voluminous data itself. This exact process happens, for example, in the SSL/TLS protocol.

Even though encryption is a common use case of public key cryptography, another common application of it is called *digital signature*, which is a cryptographic process that guarantees the property of *non-repudiation* — the assurance that someone cannot deny the validity of something (for example, if someone manually signs a contract, he/she cannot deny it in the future, since its signature would prove it wrong)([22]).

Digital signature describes a process where someone signs some data with his/her private key, and this signature can then be verified by anyone by using the public key. In other words, the logic of digital signatures is basically the reverse process of public key encryption (as it is with RSA, for instance): instead of encrypting the information with the public key and decrypting it with the private key, it is actually encrypted (signed) with the private key and decrypted (verified) with the public key. Digital signatures can then be used like real world signatures, and thus can be really useful to many applications, such as electronic commerce.

The four public key algorithms that can be listed as the most popular ones are ([23], [24], [25], [26]):

- *Rivest-Shamir-Adleman (RSA)* — the most popular public key algorithm, which is based on the problem of factoring. It can be used for both data encryption and digital signatures;
- *Elliptic Curve Digital Signature Algorithm (ECDSA)* — based on elliptic curves, instead of factoring (RSA) or logarithm functions (Diffie-Hellman). Because of that, it can reach similar security strength with shorter keys, compared to RSA and Diffie-Hellman;
- *Diffie-Hellman Key Exchange* — not exactly an algorithm for encryption and decryption, but, as the name suggests, it is intended for the exchange of symmetric keys in an insecure communication channel;
- *Digital Signature Algorithm (DSA)/Digital Signature Standard (DSS)* — developed by Digital Signature Algorithm (DSA), this algorithm is optimized for digital signatures, even though some of its implementations allow encryption and decryption.

Note: For details about these algorithms mentioned above and possible attacks that can be performed on them, see [7], [21], [12], [8].

3.3.2 Digital Certificates and Public Key Infrastructure (PKI)

Now that public key encryption has already been introduced, we can move forward to the definition of *digital certificates* and *public key infrastructure*. Digital certificates are basically documents that associate a specific entity — which may be a person, a server, a company, and so on — with a public key ([27], [28]).

The format of digital certificates that is most widely used nowadays is x.509 (RFC 5280). This standard defines some fields such as ([29]):

- X.509 version;
- Serial number of the certificate;
- Algorithm that the issuer used to sign this certificate;
- Issuer name — name of the entity that issued this certificate, which use to be a Certificate Authority (CA);
- Validity period (from when it starts to when it ends);
- Subject name — name of the entity that owns this certificate;

- Subject public key information — both the public key itself and the public key algorithm;
- Some other optional fields, such as Issuer Unique Identifier (only available in X.509 v2), Subject Unique Identifier (only available in X.509 v2) and the Extensions (only available in X.509 v3).

As well as real life official documents, such as passports, driver licenses and official IDs, have their legitimacy and validity based on the fact that they were issued by a publicly known and trusted authority (government institutions, for instance) — which can be verified by some specific watermark, signature or special stamp on the document —, so do digital certificates. To be more specific, the digital certificates legitimacy and validity are given by *public key infrastructure* and *public key encryption* (by the usage of *digital signatures*).

As just stated, digital certificates work very much like those official documents. These certificates are issued (usually under payment) by the so called Certificate Authorities (CAs), which bind the identity of a user or system to a public key, with a digital signature ([30]). These CAs themselves are entities that, as well, own certificates that are issued and signed by other Certificate Authorities. This chain continues until reaching the root Certificate Authority, which has a self-signed certificate, and should be trusted and, preferably, publicly known and accepted.

Having this said, in order to check whether a certificate is valid — and has not been forged, for example —, all this chain of CAs is traversed, verifying the validity and the signature (by using their respective public keys) of every certificate, until reaching the trusted root CA and checking its signature as well (see Figure 3.4). Moreover, the Certificate Revocation List (CRL) of each Certificate Authority in the chain is also verified, in order to see if any of the certificates has been revoked.

Finally, Public Key Infrastructure (PKI) can be described as a set of components that, together, can make the whole architecture of digital certificates work. The following list shows the components that form a PKI ([30]):

- *Certificate Authority (CA)* — issues certificates and publishes Certificate Revocation Lists (CRLs);
- *Registration Authority (RA)* — works as an interface between the user and the CA, by authenticating the user and sending the certificate request to the CA;
- *Certificate Repository* — a place to stow certificates, keys and CRLs, and make them available;

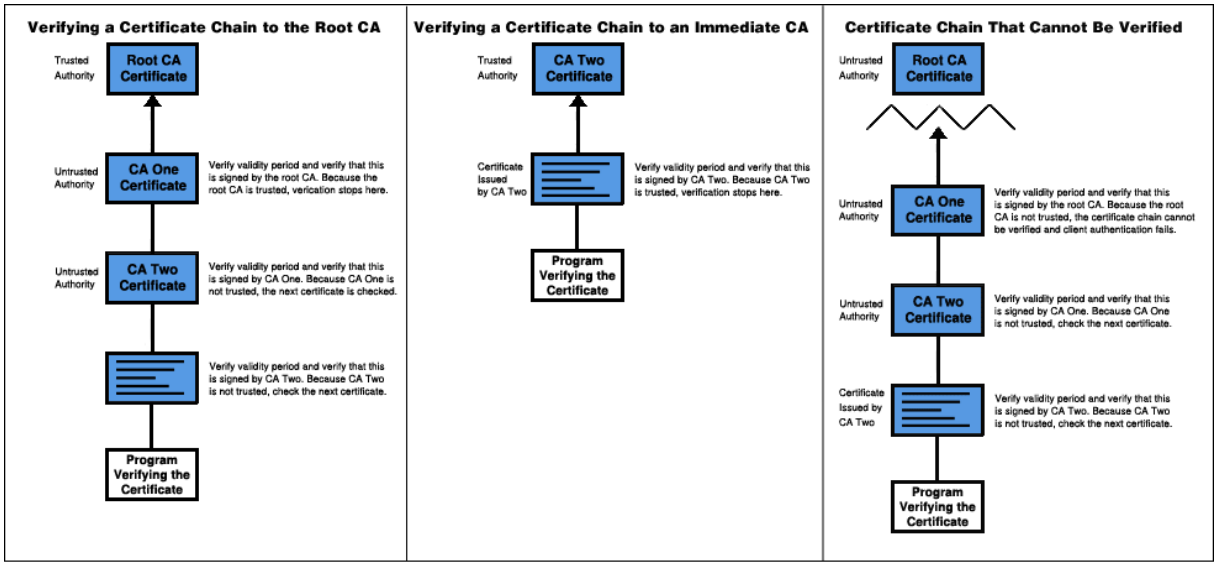

Figure 3.4: Diagrams with little explanations about some cases of *Certificate Chain Validation*. (Images taken from IBM Knowledge Center).

- *Security Policy* — security specifications, such as the verification process of the certificates, how CAs will work, how keys will be generated and so on;
- *Applications that make use of digital certificates* — for example, electronic commerce, digital banking, cryptographic protocols such as Secure Socket Layer (SSL), digitally signing files, and so on.

Digital certificates and Public Key Infrastructure (PKI) can together achieve three security properties: **integrity, non-repudiation and authentication**. As observed before, public key cryptography alone could achieve only the first two security functions, but, when working along with digital certificates and PKI, authentication is also made possible.

Chapter 4

The Security Architecture of AngraDB

In this chapter, it will be seen in details what exactly was missing in terms of security in the project AngraDB, everything that was thought in order to fill these blanks, and how they were designed to do so. In other words, in this chapter, it will be presented the security requirements that were needed in AngraDB, and it will also be explained, in a top level perspective, the solution for these requirements, which are basically the reason of all this work. Finally, some attack models will be proposed and discussed, in order to evaluate, in a subjective way, the new AngraDB security design.

First, it is necessary to understand the situation that AngraDB was in. As mentioned a few times before, in regards to security concerns, there was nothing thought nor implemented yet.

The following metaphor can be useful to help you understand **part** of the security context of AngraDB: imagine a company that is intended to hold people's belongings, however, this place where the company stores all the objects does not have any walls, fences, nor security crew, nothing! A place like this can be easily rigged by any malicious party in many different ways: the storage, in the best scenario, can simply be snooped, but it can also be stolen, or even sabotaged. That same way was AngraDB. Duo to the lack of security mechanisms, anyone could connect to an AngraDB server and manipulate anything as wanted, from reading documents in a database to deleting whole databases.

The other part that was missing in this metaphor is the fact that AngraDB also did not use any protocols to protect the communication channel — which is the TCP/IP scheme, in our case — until then.

Now that the overview on what was missing in terms of security has just been shown, it is now possible to talk about each security requirement of AngraDB individually — authentication, authorization and transport layer security —, conceptualizing and detailing

each of them, besides showing and comparing the way other great database management systems, such as MongoDB and CouchDB, fulfilled these same necessities in security, and, lastly, discussing attack models that can be applied to these security subjects.

Before diving into each requirement, it is also very important to remember that AngraDB is a project that has a huge focus on **flexibility**, and this means that, more than simply implanting these security requirements, they need to be designed carefully, so that other security modules, such as another authentication module, can be attached to AngraDB and used with ease.

4.1 Authentication

4.1.1 The Requirement

It is very common to see people confusing the concepts of authentication and authorization, so, first, let us make them clear once and for all. While **authorization** regards to what actions an user can or can not take inside an application (more on this later, in the authorization section), **authentication** takes care of identifying uniquely each user that accesses this very application.

Identifying uniquely a user is important because, once the application now knows who the client currently connected is, it is now possible, for example, to audit every action of this user, or even manage his/her permissions upon the application (his/her authorization).

This second example is crucial, because we have just established that, for **authorization** to exist, it is first necessary that an **authentication** scheme exists. In other words, in order to be able to set and manage someone's permissions in some application, this application first needs to label this client as an unique user, so that this very user can have his accesses managed. The project AngraDB, as well as many other applications, being databases or not, has the same need of an authentication scheme in order to be able to establish later an authorization mechanism, and hence, keep control of the actions of each user, which is actually the whole point of authentication and authorization when combined.

It is worth mentioning that, in order to exemplify the flexibility of AngraDB's security interface, the authentication requirement will be contemplated with two mechanisms, different than the other two security requirements, which had only one implemented mechanism each.

4.1.2 Authentication Mechanisms

When building an authentication scheme, there are some options of authentication mechanisms to choose, but two of them are the most known: **challenge-response mechanism** and **authentication with digital certificates**. Even though there are many variations of these mechanisms, these two can still be considered the most used top level designs when the subject is "authentication".

The challenge-response mechanism is by far the most popular. As the name suggests, it refers to an authentication system that, in order to be successfully authenticated, the user will have to answer a specific question correctly, that is, he/she will have to give the correct **response** when presented with a **challenge** by the application (see Figure 4.1). Usually, this challenge is simply a password, which the user has to send along with his/her username or any other unique identifier that is established by the application.

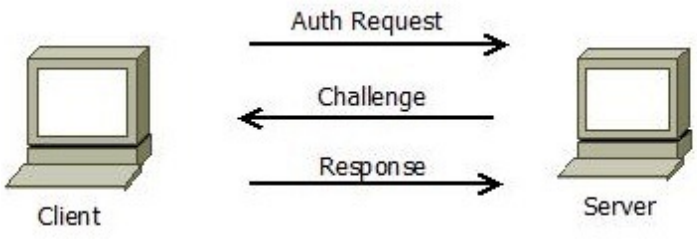

Figure 4.1: Simple challenge-response based authentication (image taken from `http://www.defenceindepth.net/2011/04/attacking-lmntlmv1-challengeresponse.html`).

This mechanism is used on both Apache CouchDB and MongoDB (which can also be configured to use authentication based on digital certificates), even though they differ a little bit on some setups of this mechanism and on some pieces of its architecture.

Authentication with digital certificates also could not be more straight forward. Basically, to identify uniquely the users securely, this authentication mechanism counts on the the reliability of digital certificates given by a Public Key Infrastructure (PKI), formed by publicly known and trusted Certificate Authorities (CAs). Usually, all an user needs to do to get authenticated using this method is present his/her digital certificate and its private key, which may either be inside a file or not (on Figure 4.2, it is possible to see a good and common example of certificate-based authentication).

4.1.3 Challenge-Response Authentication

Now that the core concept of the challenge-response mechanism has already been presented, we can approach some deeper details about, for example, how the architecture of

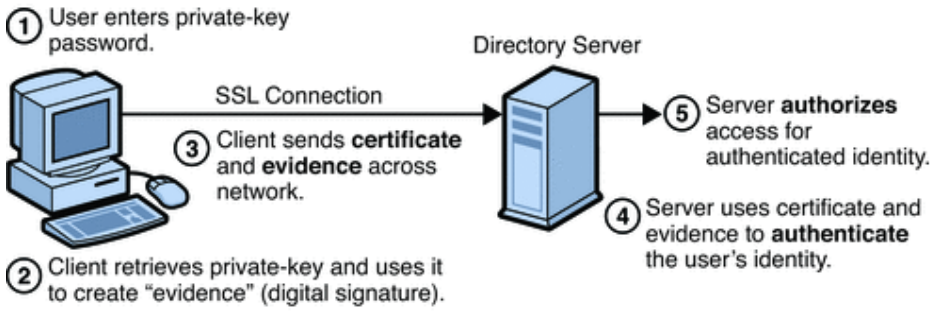

Figure 4.2: Basic example of a digital certificate-based authentication scheme (image taken from `https://docs.oracle.com/cd/E19575-01/820-2765/6nebir7eb/index.html`).

a scheme of this kind is formed. Furthermore, once shown the details of this authentication scheme, we will also present some comparisons between the ways MongoDB and CouchDB use challenge-response in authentication.

Even showing the details, this mechanism still looks very simple. A more thorough step-by-step about how an authentication scheme of this kind works would look like this:

1. Client requests authentication to the server;
2. Server sends the challenge to the client. **Observation:** In some cases, step 2 and step 3 may be encompassed in step 1, which would then become: *Client requests authentication to the server, sending his credentials as well*;
3. Client sends the response back to the server;
4. Server checks whether the client's response matches to the password that is stored in the authentication database for this user (if the user even exists);
    - Usually, before being stored in the authentication database and also before being checked in this very database, the password may pass through a cryptographic hash function, in order to make the process of obtaining someone's password more difficult, in case this authentication database is leaked or invaded;
    - Furthermore, some real world applications even append a "salt", which is a pseudo-random string of data, to the passwords before hashing them. This step is also to avoid some attacks on a leaked or invaded passwords database (more on these attacks later, in the Attack Models chapter).

5. If the response given by the client matches with the one that was previously stored in the authentication database, the server changes the status of this user to "authenticated" and tells the client that the authentication process occurred successfully. Otherwise, in case this user did not even exist in the database yet, or if the client's response did not match, the "not authenticated" status is maintained, and the server sends a message back to the client, saying that the authentication process failed.

These steps above are reproduced in a great part of authentication schemes that are based in the challenge-response mechanism. However, each application may differ a little bit in some of aspects, but basically, some of the main differences between them lies in how the password is hashed before getting stored in the database — that is, what algorithm(s) is(are) used for password hashing — and how the authentication information of the users is stored.

MongoDB's and Apache CouchDB's Challenge-Response Mechanisms

As just said, the main difference between the Authentication architectures of these DBMSs lie in the way users' authentication information is dealt with and stored, and also how the users' passwords are manipulated and hashed before going to the database.

Regarding to the first point, MongoDB's way to store authentication information might seem a little confusing at first sight. In its documentation, it is stated that, when a user is created in a specific database, this will be its *Authentication Database*. Some people may think that, hence, this user's authentication information will be stowed there, but, actually, all the authentication data is stored centrally, in a collection (*system.users*) inside *admin* database. So, the real function of the *Authentication Database* is serve, along with the username, as a unique identifier for a user — which means that users with the same username can be created in different databases, since its identity is given by the username **and** its authentication database together. CouchDB's way to deal with authentication data looks simpler because it just uses the username as unique identifier, as it is normally done, and store all the user's authentication information in a central authentication database as well.

In regard to how these DBMSs use to manipulate and hash the passwords when using the Challenge-Response mechanism for Authentication, MongoDB uses what they call Salted Challenge Response Authentication Mechanism (SCRAM), which is based on RFC 5082, the stardard that defines best-practices on challenge-response mechanisms, and CouchDB goes with PBKDF2, which is rather based on RFC 2898.

Even though the names are different, in practice, they offer almost the same: the MongoDB's SCRAM features different random salt per user and an adjustable iteration count,

as well as CouchDB does while using PBKDF2. Thus, the differences lie basically on the fact that MongoDB's SCRAM performs a mutual authentication (server to the client and client to the server), while that does not happen with CouchDB's mechanism (which is only one-way authentication: client to server), and also on the fact that PBKDF2 is much more liberal on the hash function — it actually expects pseudo-random functions as parameter, which is a much broader concept — comparing to SCRAM, which only accepts SHA-1 and SHA-256 as hash functions. Although it was not clear in their documentation, CouchDB apparently uses SHA-1 as the hash function inside PBKDF2.

Note: For more information about how the authentication mechanisms of MongoDB and CouchDB work, please check [31] and [32], respectively.

AngraDB's Challenge-Response Mechanism

We decided to follow Apache CouchDB's way to deal with authentication data because of its general simplicity, and, thus, we chose to use only the username as unique identifier, and store all the users' information in a centralized authentication database.

Now, in regards to the way the password is hashed before being manipulated in the authentication database, we have only followed CouchDB until a certain point. PBKDF2 was the chosen algorithm for key-stretching both because of CouchDB and also — and mostly — because of its popularity among the community when the matter is *password hashing*.

However, Apache CouchDB does not enter much in details about the parameters used in PBKDF2, and thus, our decisions regarding to this are all based in a mix of researches and testing indeed different parameters on AngraDB Core. The following list discuss the decisions took for the PBKDF2 parameters:

- *hash function* — the hash algorithm that was chosen to be used inside PBKDF2 was SHA-512, since, as discussed in the Cyptographic Hash Functions section of the previous chapter, it is a mature and widely used hash function, besides not having still any concrete vulnerabilities. Moreover, it is commonly used by the community along with PBKDF2.
- *salt* — while MongoDB does not specify in its documentation how it generates each salt per user, CouchDB uses 128-bit UUIDs (Universally unique identifiers) as salts. Well, the salt needs be long enough so that it does not repeat itself between users, and thus prevent *Rainbow Table* attacks, but even if it repeats once, still it should not be a major security concern. RFC 2898, which is the PBKDF2 standard, suggests a minimum salt length of 64 bits, and the community tend to suggest 128-bit salts. Having this is mind, AngraDB opted to generate random salts formed by

22 alphanumeric characters (for sake of simplicity in implementation), which gives a number of possibilities similar to a 128-bit random salt ($62^{22} \sim 2^{128}$).

- *iteration count* — as explained before, this number dictates how many times the hash function will be executed until the final hash value is found. Its objective is to slow down its calcultion, to prevent *Brute Force* attacks. To obtain a value for the iteration count for AngraDB, some different values were tested, and the duration of authentication process for each value were compared. Without further ado, the chosen number for the iteration count was 150,000. While testing and timing some log in attempts, it took around $1.92s \pm 0.05s$ for the PBKDF2 algorithm to calculate each final digest with these parameters. (*Disclaimer: The computer used to run this test had an i5-6600K processor and a nVidia GTX 970 video card, and was running Ubuntu 64-bits in a virtual machine with 8GB of RAM available*)

The Figure 4.3 summarizes these top-level decisions about how AngraDB's challenge-response mechanism stows user's information and how it hashes users's passwords.

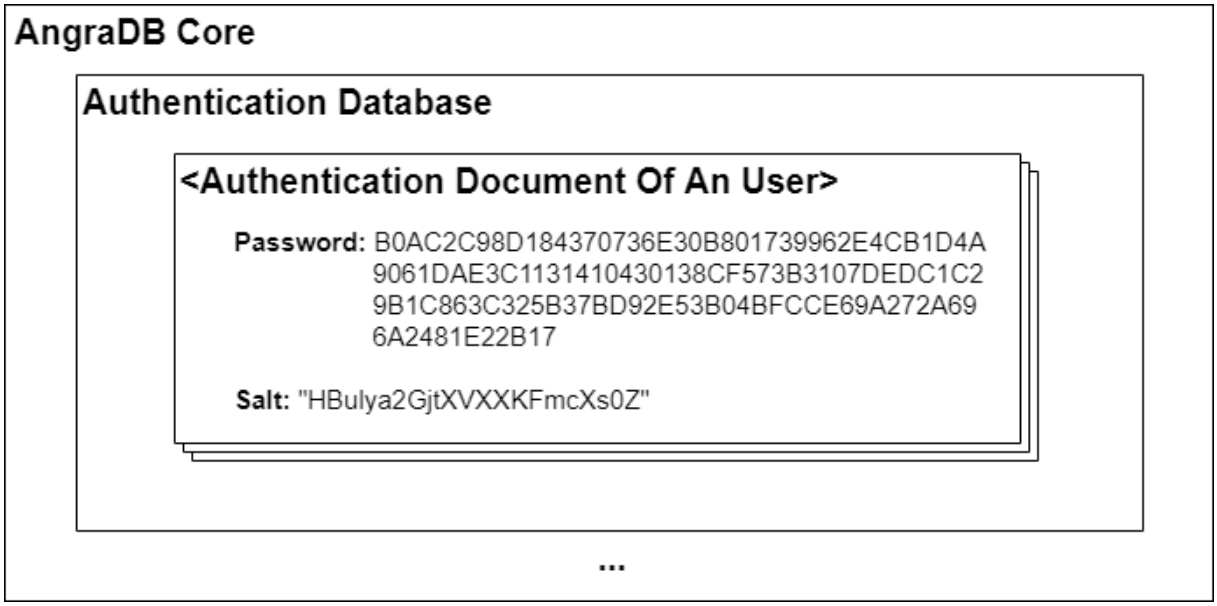

Figure 4.3: A summary/example of AngraDB's challenge-response authentication mechanism. In this picture, the password field is the hexadecimal result of an execution of PBKDF2. The input was a concatenation of the password "Password!123" and the salt present on the image, which has 22 alphanumeric characters. PBKDF2 was run with the following parameters: SHA-512 as the hash function, 150,000 iterations, and a 512-bit output.

4.1.4 Certificate-based Authentication

Note: Before reading this subsection, it is important that you first have a background on Transport Layer Security and on its handshake. This subject is approached in section 4.3, which is a section that is reserved entirely for this matter.

Even though authentication with digital certificates may seem to be complicated, it is actually quite simple. Its simplicity comes from the fact that the SSL/TLS handshake takes care of the authentication process itself — and that is why it is good to have a background on this matter first.

In the beginning of the SSL/TLS handshake, the server, while sending its certificate, may also require the client to send his/her certificate as well — which is a process called *mutual SSL authentication*, or *two-way SSL authentication*. By doing that, not only the server authenticates to the client, but the client needs to authenticate to the server as well.

As observed in Figure 4.2 and as detailed in subsection 4.3.2, in course of the SSL/TLS handshake, after the client sends its certificate and the evidence (a random byte string that is unique to this current handshake, and is known by both client and sever), which is to be encrypted with the client's private key, the authentication of the client will happen in two steps: the validation of the client's certificate itself, and the verification of the evidence, which will be performed using the client's public key.

The first step is done by verifying the certificate's expiration date and by performing the *Certificate Chain Validation*, which is a process that checks the digital signature present on the certificate using the public key of the CA that issued it. This process is repeated with the certificate of the CA and the other CA that issued it as well and so on, until reaching the root CA, which should be public known and trusted — or at least trusted by the party that is performing the *Certificate Chain Validation* (see Figure 3.4). In some cases, the *Certificate Revocation List (CRL)* of each CA is also verified, to make sure that these certificates were not revoked in this mean time.

The second step is intended to confirm if the client is indeed the owner of the certificate, after all, the client can just be pretending to be the owner, when presenting this very certificate. As it allegedly encrypted the evidence with its private key, then it should be possible to decrypt and verify the evidence using the public key found in the certificate (which was already validated in the first step). If the verification of the evidence is successful, it means that the client indeed has the pair of keys, which then means that he/she is indeed the owner of the certificate, since it should be infeasible to forge a private key for the public key that is on the certificate.

AngraDB's Certificate-Based Authentication Mechanism

Once the SSL/TLS handshake has already finished and guaranteed the client authentication, all an application that already has support for SSL/TLS needs to do is stow the desired authentication information in its authentication database, for example, the user name, the client's certificate itself, and extra data, such as the user's address, and other things that the application may find necessary. Therefore, in future authentications of this very user, the only thing to do is retrieve the user's register from the authentication database.

That was the exact approach of AngraDB in its certificate-based authentication mechanism. In this scheme, the only user information that is stored is the user name and the its certificate, which was used in its authentication. This information is stowed in the same way as it is with AngraDB's Challenge-Response mechanism, represented in Figure 4.3: there is a document in the authentication database for each user, and all the information is put there. However, as just stated, instead of the password and salt, the user name and the certificate are the data to be stored.

Note: In this subsection, it will not be presented how MongoDB and CouchDB designs the solution for Certificate-Based Authentication, since CouchDB does not even has support for this, and MongoDB does not go much in details about its architecture for this mechanism. Instead, it just explains how to authenticate with x509 certificates.

4.2 Authorization

4.2.1 The Requirement

As previously stated, it was first necessary to have established who is who inside the application, that is, it is necessary to have users identified uniquely, in order to, then, be able to specify what some user can or cannot do. Once *Authentication* is clarified, we are now able to get into *Authorization*.

Authorization, which is often called *Access Control*, plays a very important role in security: it takes care of controlling what actions an user can perform upon determined resources. It is crucial because, without *Authorization*, an application turns into chaos: either everything or nothing is permitted — being the first option more common in an application without an authorization scheme.

Going straight to the point, now that AngraDB already resolved *Authentication*, it then needs a mechanism that can discriminate each user, restricting their actions upon resources inside AngraDB. This mechanism should also permit the manipulation of the

accesses of each user. For sake of completeness, it should satisfy these following properties, as stated by [33]:

- "*Prevent access* — in the absence of any privilege, ensure that the subject cannot access the object. The principle of Failsafe Defaults says that this should be the default.
- *Determine access* — decide whether a subject has access, according to some policy, to take an action with an object.
- *Grant access* — give a subject access to an object. The principle of Separation of Privilege says this should be fine-grained; don't grant access to many objects just to enable access to one.
- *Revoke access* — remove a subject's access to an object.
- *Audit access* — Determine which subjects can access an object, or which objects a subject can access."

4.2.2 Access Control Models

In order to guarantee those points above mentioned, achieving that by establishing concepts and relations between subjects (the user, or process), objects (the resource itself, like a table or a collection) and operations (such as read and write), there are some models that can be used as base for the implementation of the desired *Authorization* scheme. These are some of the many and many existing *Access Control Models*:

1. Discretionary Access Control (DAC);
2. Role-Based Access Control (RBAC);
3. Mandatory Access Control (MAC);
4. Attribute-based Access Control (ABAC);
5. Rule-Based Access Control (RAC).

Because there are lots of different models, only the first two will be approached in this work: Discretionary Access Control (DAC) and Role-Based Access Control (RBAC).

Discretionary Access Control (DAC)

The Discretionary Access Control model ([4], [33], [34]), among those three mentioned, is probably the simplest one, and also one of the most widely used as well. When using

this model, the owners of resources can tell what permissions or what operations can be performed by each specific user upon each specific resource.

This information, which correlates the resources and the subjects, telling what operations they can perform, can be represented by the so called *access control matrix*, introduced by Lampson in 1974. An example of this very matrix can be observed in Figure 4.4.

	Medical record	*Administrative record*	*Prescriptions*
Alice	**W,R**	**R**	**R**
Bob		**R**	
Charly	**W,R**	**W,R**	**R**
David	**R**	**R**	

Figure 4.4: Example of a simple *access control matrix*. Inside the cells, "W" are equivalent to "Write" operation, and "R" to "Read" operation. As said before, the absence of privilege — that is, empty cell — means no access at all. (Table taken from [4]).

Even though this table is good to visualize all the access information, applications do not use to implement it as it is. There are actually two common ways to store this information: either by rows or by columns.

The first one is normally called *privileges list* or *capabilities list*, and it basically shows which permissions **a determined subject** has upon some resources. Taking the Figure 4.4 for example, it would be something like: *David — read access upon Medical Record and Administrative Record.*

The second one though, which is the most popular, is called *access control list (ACL)*. It organizes the permissions of subjects **per resource**. For instance, having the Figure 4.4 as example again: *Medical Record — Alice and Charly can perform Read and Write operations, and David can only perform Read ones.*

Role-Based Access Control (RBAC)

"A role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on a member of the role" — Sandhu, Coyne, Feinstein, and Youman (1996) ([4]).

Role-Based Access Control model ([35], [4], [34]) organizes accesses based on roles that groups of subjects play in some organization, or, in our case, in the application. In other words, when using RBAC, the permissions are not given to the subjects directly, as

it is with Discretionary Access Control; instead, they are given to roles, which are then associated to the subjects. In this case, each user — or subject — will have permissions according to the role associated to him/her, as observed in Figure 4.5.

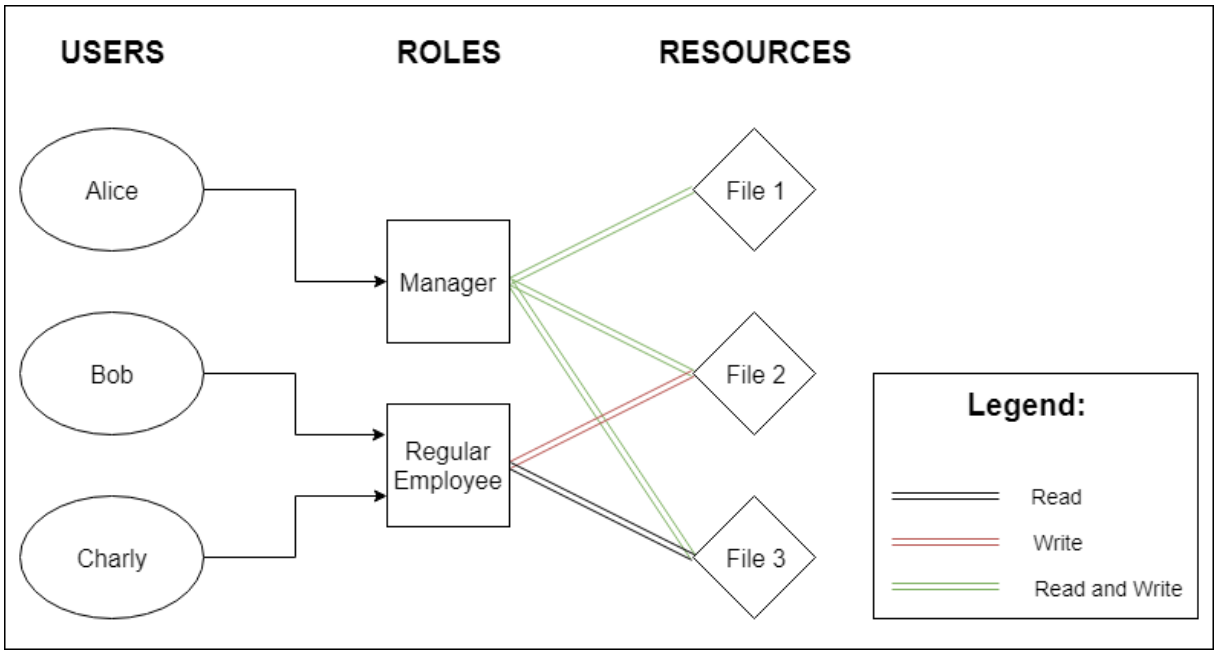

Figure 4.5: Illustration of a simple Role-Based Access Control (RBAC) model.

Even though this feature may not be present in every application that uses authorization based on roles, RBAC specifies that **roles are hierarchical**, which means that a role should be able to inherit from another role, earning, hence, all the permissions from its parent.

It is worth mentioning that, regardless of the chosen authorization scheme, the application that utilizes it should follow the *Principle of Least Privilege (PoLP)*, which is a concept that says that an user should only have the necessary permissions to perform his/her intended actions ([35]).

4.2.3 MongoDB's and CouchDB's Access Control Systems

Both of these Database Management Systems use the Role-Based Access Control model. However, the Apache CouchDB's implementation and usage is much more basic than MongoDB's, which is more robust and thorough.

CouchDB's Access Control System

CouchDB's authorization scheme is actually quite lean. It comes only with two built-in classes (which are applied in database-level): member and admin. Users that are members

can read, edit and create documents inside the database, except the ones they call *design* documents. Admins, thus, are able to create, read and modify any type of document, besides configuring other users' roles and the database itself.

Note: It is important to make explicit that there are two kinds of admins in CouchDB: the database-level admins and the server-level admins.

Besides that, custom roles can be created by an admin. At first, it might look strange that these roles can only be assigned to those two classes that were just presented. What if someone wanted to create a role that has just read-only permission, or anything that is more customized than what "member" and "admin" classes can offer? Well, there is nothing built-in that facilitates this process. Instead, it is only possible by implementing authorization functions that are run before each user action. These functions will tell whether the user, based on his roles, will be able or not to take his/her desired action.

MongoDB's Access Control System

On the other hand, MongoDB has everything already built-in. As specified in its documentation, MongoDB treats roles as a group of privileges, being each privilege composed by a resource and the permitted operations upon this resource ("operations" are referred to as "actions" in MongoDB's documentation).

Dealing with roles in MongoDB really looks very straightforward. For starters, it already comes with a wide variety of built-in roles, which are split in a few categories:

- Database User Roles — regular roles with privileges that apply to single databases: *read* and *readWrite*;
- Database Administration Roles — administration roles with privileges that apply to single databases: *dbAdmin*, *dbOwner* and *userAdmin*;
- All-Database Roles — roles with privileges that apply to all databases (except for the ones they call the *local* and *config* databases): *readAnyDatabase*, *readWriteAnyDatabase*, *userAdminAnyDatabase* and *dbAdminAnyDatabase*;
- Superuser Roles — this role provides full privileges to all databases: *root*;
- There are also Cluster Administration Roles, Backup and Restoration Roles and Internal Roles.

To create new custom roles in MongoDB is quite trivial as well. Given the fact that roles are a group of privileges, and that each privilege is made of a resource and its permitted actions, all that is needed in order to create a new role is compose it with your desired privileges. As "resource", clusters, full databases or collections inside

a database can be chosen. For the actions, MongoDB has a list of operations grouped by common purpose, and you can choose whichever actions you want to compose your desired privileges.

It is important to mention that roles in MongoDB are hierarchical! Hence, if specified that a role inherits from another role, this very role will keep all privileges from its parent, as explained before.

4.2.4 AngraDB's Access Control System

Even though we have been using MongoDB and Apache CouchDB as parameter to build our needed security requirements' architectures, it will not be the case with *Authorization*.

As one of the biggest challenges with this work was actually designing and building the generic interfaces themselves so that people could later attach other security modules according to their will, we opted for the simple, regarding to the architecture of the Authorization mechanism itself.

Instead of opting for Role-Based Access Control (RBAC), as MongoDB and CouchDB did, the access control model that was chosen for AngraDB in this work was Discretionary Access Control (DAC), which is also a very popular model, as discussed before.

In relation to the storage of authorization information, it is done in the *access control list (ACL)* format: there is a document inside the authorization database for each database in AngraDB, and, inside this document, there is a list of users and their permissions upon this database that this document is associated to.

In AngraDB's DAC, there are four possible permissions to be chosen, which are applied in **database level**, that is, these permissions are valid for a database and all resources available inside it. Check the list of permissions:

0. *NoPermission* — As the name suggests, if a subject has this permission assigned to him in a database (or if the subject has no authorization information at all), then he/she has no access to this database;

1. *ReadPermission* — Subjects with this permission can perform read operations, such as look-ups and query terms on documents inside this database;

2. *ReadAndWritePermission* — Subjects with this permission are able to execute read and also write operations, including the creation of new documents, and edition and deletion of existing documents inside the database;

3. *OwnerPermission* — This permission allows a subject to do everything the other permissions already do, and also enables a subject to delete the database and ma-

nipulate other users' permissions inside this very database as well, by editing or granting some permission, or even listing all of them.

It was possible to observe that these permissions work like concentric circles: from a permission to a higher one, the higher one is able to do everything the other can, and is also able to perform some more actions. This is done like this because, when these permissions are stored in the Authorization database, they are stowed as integers (the same integers that can be noted in the list above). That way, when a permission for certain operation is requested, its number is compared to the one in the database, related to the subject; if the subject's number is equal or greater, then the access is allowed, and, otherwise, denied. An overview on how this authorization information is stored can be checked on Figure 4.6.

Before the execution of every subject's action, the permission needed for this operation is verified, using the process that was mentioned above to decide whether the action can be taken or not. It is important to mention that some operations inside AngraDB do not need any permission to be performed (actually, they require, as minimum permission, "*NoPermission*", which is the same as saying that it does not require any permission), such as registering, logging in and out, and creating new databases (of which you will receive "*OwnerPermission*" when doing so).

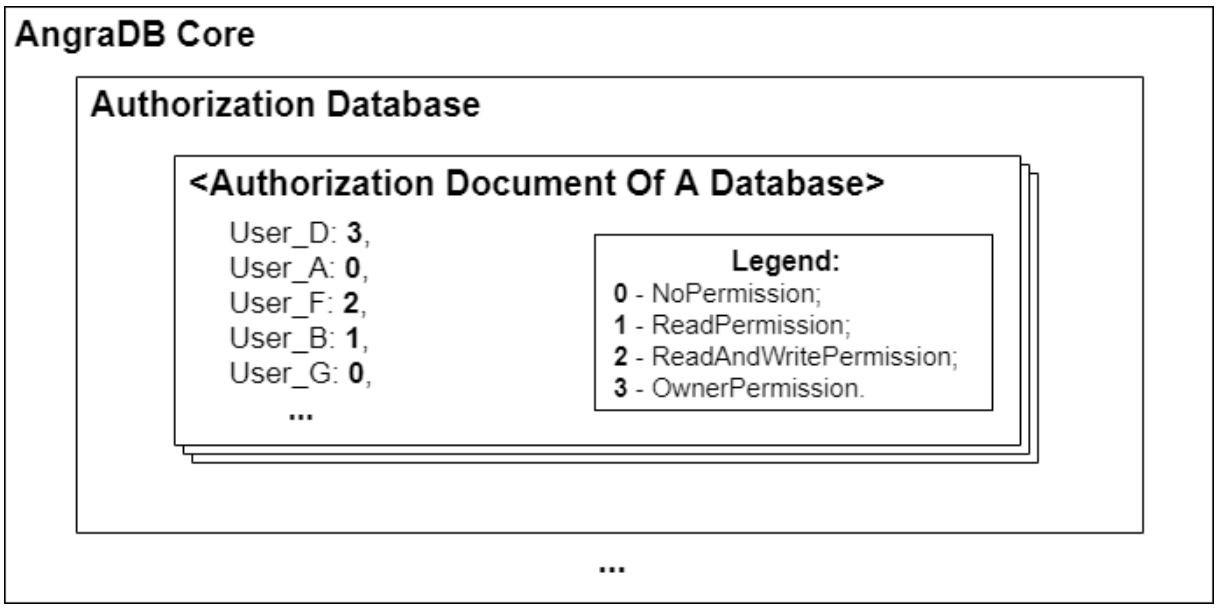

Figure 4.6: Illustration of the storage architecture of AngraDB's authorization scheme.

4.3 Transport Layer Security

4.3.1 The Requirement

We have stated that *Authorization* needs *Authentication* to work, but there is still a problem in this scenario. With what was discussed until this point, the operations inside the application are more restrictive, since we already have an *Authorization* system that is working along with an *Authentication* mechanism (or mechanisms). The problem is that, in order to be able to access the application and perform the desired actions, it is first necessary to send sensitive information — the user's credentials — to the server at the time of authentication.

As explained before, AngraDB uses Transmission Control Protocol (TCP) to communicate externally, and given the fact that there were no security mechanisms protecting the communication medium, it is possible to easily capture all the packets that are traded between the server and the client and see all the information that is being traded **in plaintext**, using, for example, some network traffic analyzer, such as Wireshark or tcpdump, as can be observed in Figure 4.7.

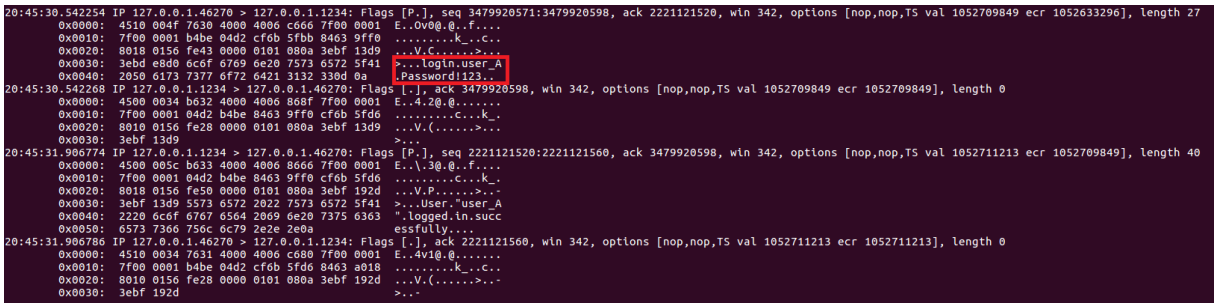

Figure 4.7: Screen-shot of packets that were captured using *tcpdump* in Ubuntu, right after a log in attempt in AngraDB. As it is possible to see, the user's credentials can be easily noted in the payload of the packet (user name: *user_A*, and password: *Password!123*) .

Having this said, AngraDB needs a way of ensuring secrecy when communicating externally, that is, a way to trade information in an encrypted channel, making it infeasible for eavesdroppers to reach and tamper the actual content that is being traded.

To fulfill this blank, there is a protocol whose name is very self explanatory, even though not so creative: the Transport Layer Security (TLS) protocol, successor of the deprecated Secure Socket Layer (SSL) protocol.

4.3.2 The SSL/TLS Protocol

SSL/TLS emerged exactly from the necessity of creating an encrypted communication channel that could be secure and have secrecy, preventing adversaries from eavesdropping and tampering its content.

As a historical background ([36]), Secure Socket Layer (SSL) was developed by Netscape, but its first version was never published publicly. Its version 2.0 was published in 1995, but it was quickly replaced by version 3.0 in 1996, since a variety of vulnerabilities were being discovered and exposed. Transport Layer Security (TLS) is basically another version upgrade of SSL, since it is practically SSL v3.0 with some improvements and fixes on some other vulnerabilities. Is short, both SSL and TLS refer to the same protocol itself, being TLS essentially SSL v4.0, but with a different name, even though a great part of the community still calls it SSL.

SSL/TLS is a protocol that operates above the transport layer, and is a clear example of both public-key encryption and symmetric-key encryption. It basically establishes a handshake whose objective is to, first, confirm the identity of one or both parties (e.g. a client, which may be a browser, and a server), and then share information about how the channel will be encrypted from then on — this information is essentially the encryption key and the encryption scheme that will be used ([37], [38], [36]).

Here is an overview on the SSL/TLS handshake (which was built gathering information from [39], [40], [41], [38] and [37]):

1. Client sends a *Client Hello* message, with his/her SSL/TLS version, the cipher suites that are supported by the client (sorted according to client's preference) and a random byte string that will be used in subsequent computations, which we will call *client_random*;

2. Server first verifies the client's SSL/TLS version, and then sends a *Server Hello* message, with the cipher suit that was chosen from the client's list, the session ID, another random byte string (*server_random*), and, most importantly, the Server Certificate (which contains the server's public key);

3. *OPTIONAL* — The server may require the client's certificate as well. If this is the case, the server also sends, along with the *client's certificate request*, a list with the supported types of certificate and acceptable Certification Authorities (CAs);

4. The client authenticates the server by validating its certificate, which was just sent by the server. If the authentication fails, a warning may be emitted or the handshake may fail. Then, the client generates a *pre_master_secret*, encrypts it with the

server's public key (which was inside the server's certificate) and sends it back to the server;

5. *OPTIONAL* — If the server had previously requested the client's certificate, then the client sends, along with what was said to be sent in step 4, his/her certificate and an encrypted random byte string that is known between both parties (*client_random* or *server_random*, for example), which is encrypted using the client's private key so that the server can use it to authenticate back the client with the client's public key;

6. *OPTIONAL* — If the server had previously requested the client's certificate, then it authenticates the client by verifying its certificate and by decrypting and verifying the data that the client sent. If the authentication fails, the handshake fails as well;

7. The server then decrypts the *pre_master_secret* with its private key (this is an important part of the server authentication process as well, since the server party will only be able to decrypt the *pre_master_secret* if he/she is indeed the owner of the pair of keys, and hence, the owner of the server certificate). Now that both the server and the client have the three pieces (*client_random*, *server_random* and *pre_master_secret*), they **both** will use them to generate the **session key**, which is a symmetric key that will be used to encrypt and verify the integrity of the data that will be traded between the server and the client along the session.

8. The client tells the server that the incoming messages will be encrypted using the session key, and also sends an encrypted message saying that the handshake is finished on the client's behalf;

9. The server does the same and tells the client that the incoming messages will be encrypted using the session key as well, and also sends an encrypted message saying that the handshake is finished on the server's behalf. The fact that both parties have received the *Finished* message, which was encrypted using the session key, is enough for them to acknowledge that the handshake was successful, and now they can start exchanging messages securely and with secrecy.

For more technical and detailed information about the Secure Socket Layer (SSL) or the Transport Layer Security (TLS) protocols, check [37].

4.3.3 AngraDB's Transport Layer Security

Regarding to Transport Layer Security, both MongoDB and Apache CouchDB come with built-in support for SSL/TLS, and so was done with AngraDB.

SSL/TLS support was implanted in AngraDB, being now possible to choose between **pure TCP**, or **TCP with SSL/TLS** — using the versions that are supported by the Erlang SSL application: SSL-3.0, TLS-1.0, TLS-1.1, TLS-1.2, DTLS-1.0 (based on TLS-1.1), DTLS-1.2 (based on TLS-1.2), by the time this monography was written. If the second option is chosen (TCP with SSL/TLS), the only necessary thing to do is configure the AngraDB start-up settings, by setting the server's certificate information such as the server's certificate itself, the certificate key and the Certificate Authority (CA) certificates.

By configuring it as said, AngraDB is now able to listen to incoming SSL/TLS connections, perform the SSL/TLS handshake properly, and, finally, trade messages securely and with secrecy.

It is worth reinforcing that, if AngraDB is not configured to use SSL/TLS and the server's certificate information are not set, it still works like it used to before, listening to incoming pure TCP connections, executing pure TCP handshakes and exchanging messages via TCP in plaintext.

As a counterpoint to Figure 4.7, which shows all the content of packets that were captured after a login attempt in AngraDB in plaintext, including the user's credentials, now that AngraDB uses a fully encrypted communication channel, it is now possible to see, in Figure 4.8, that the packets' payloads cannot be eavesdropped anymore, since they are all encrypted.

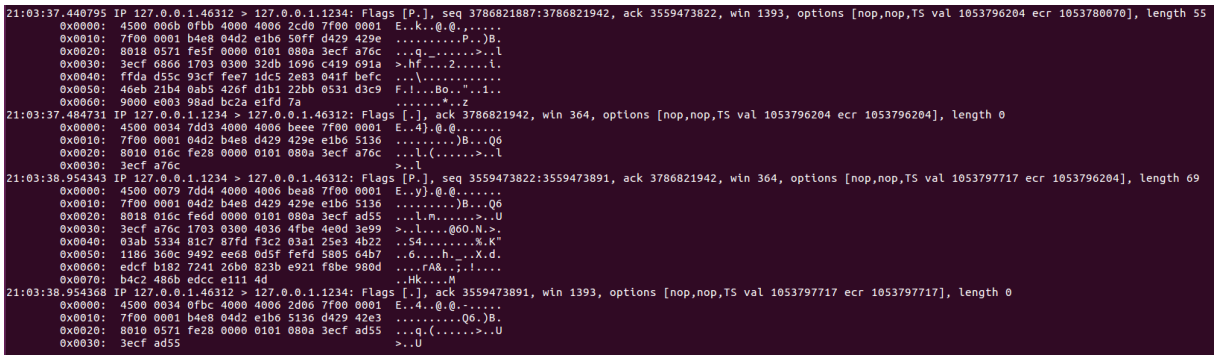

Figure 4.8: Screen-shot of packets that were captured using *tcpdump* in Ubuntu, right after a log in attempt in AngraDB, using SSL/TLS. Now that this protocol is being used, the content of the packets are encrypted and no longer readable (or, at least, no longer understandable) as they were in Figure 4.7, which means that the user's credentials are now kept in secrecy through the communication channel. (user name: *user_B*, and password: *Password!123*) .

4.4 Evaluation Based On Attack Models

As said in the introduction of this chapter, this section in intended to evaluate — subjectively — the AngraDB security design that was just proposed and discussed throughout this very chapter, by presenting some attacks and approaches used by attackers that are popular and commonly performed, and then discussing how this new AngraDB security design would deal with them.

Note: This section will focus on attacks and flaws on the challenge-response authentication mechanism, since the other features such as the certificate-based authentication and the transport layer security have their securities based on the Erlang's implementation of the TLS protocol — which already uses its most recent version (TLS v1.2).

An attacker who intends to impersonate users' identities and use their accesses to perform its malicious actions, needs to bypass the Authentication mechanisms, which are basically the first line of defense against unauthorized access.

There are some basic and common ways to try to bypass an authentication scheme, being the most part of them related to flaws on challenge-response authentication mechanisms.

The most basic attack would be a simple password guessing, where the attacker, once having someone's username, tries some passwords manually. Furthermore, this attack may also evolve a little bit and turn into a *Brute Force* or a *Dictionary* attack.

Brute Force attacks consist of attempts to guess someone's password by running a program that tries many different combinations, changing character by character, until finding the correct password.

Dictionary attacks are similar to *Brute Force* attacks, but they are more efficient, since, instead of trying many random combinations, it tries a set of common passwords (a dictionary of passwords).

Usually, there are three flaws that can make these attacks succeed:

- *using verbose messages* — when specific messages such as "incorrect password" are used, the application is giving the attacker sensitive information, since, in the case of this example, it is basically confirming that the username is valid, making it easier for the adversary;
- *allow to try indefinitely* — this permits the attacker to use all its computational power to try to find the correct password, which may result in millions or even billions of attempts per minute, with attacks such as the brute force or dictionary;
- *using a bad password* — using common or simple passwords facilitates attacks such as dictionary or even simple password-guessing, since they use a list of common

passwords — a huge list, in case of a dictionary attack — to try to guess the user's secret.

In regard to the first point, AngraDB cares about not being not overly verbose. In a case such as the exemplified, for instance, AngraDB would emit the message "incorrect username or password".

Even though AngraDB still lets users try to authenticate indeterminate times, it uses the PBKDF2 key-stretching algorithm to slow down the password hashing, making one single authentication attempt take about two seconds, which compromises the effectiveness of attacks that count on repetition (such as the brute force and the dictionary attacks).

Regarding the last point, AngraDB's challenge-response mechanism does not yet reinforce the user's password quality at the moment of registration, which may let users utilize bad passwords.

Besides attacking the challenge-response mechanism itself, there could be cases that, for example, the authentication database is leaked. Unfortunately, leakages happen, but authentication mechanisms needs to be prepared for that and have containment measures.

The worst possible scenario is an authentication database that saves the user's passwords in plaintext. In these cases, the passwords are already available for untrusted parties. However, even authentication databases with hashed passwords are susceptible to attackers.

Password hashing is literally the last security measure between the attacker and the passwords themselves, so it needs to be robust in order to prevent some specific attacks. (The discussion about features of a good password hashing scheme can be check on the Cryptography chapter, in the section 3.2).

In order to overcome the password hashing and get to the passwords, *Brute Force* and *Dictionary* attacks are still used. The difference now is that each attempted combination needs to be hashed before being compared to other hashes in the leaked database.

Besides these two attacks, there is one called *Rainbow Table* attack. This one is more efficient than the other two attacks when the matter is cracking hashed passwords. *Rainbow Table* attack is basically a big table of common passwords and their pre-computed hash values. With this table in hands, attackers can easily check if any of the pre-computed hash values on the table is present in the leaked database. If the authentication scheme does not have any plans against such attacks, it is possible that many hash values of the database match with the hash codes from the table. Once the hackers know which digests did match, they just need to check, on the table, the correspondent passwords.

As explained in the Cryptography chapter (section 3.2), the AngraDB Authentication system counts on a password hashing scheme that is strong against these three attacks (brute force, dictionary and rainbow table attacks).

Besides slowing down the calculation of the final digest (which compromises the feasibility of brute force and dictionary attacks), the key-stretching algorithm PBDKF2, used in AngraDB for password hashing, appends a pseudo-random 22-character alphanumeric *salt* to the password before hashing it 150,000 times. By concatenating this *salt* to the passwords, the pre-computed hash values on the rainbow table **are no longer useful**. Moreover, even appending a salt to the end of each possible password in the table and recomputing their digests, it would only work for a single password, since the random salts are generated per user.

Even though this security design has increased AngraDB's security substantially, it still has a lot to enhance. For example, AngraDB security system still has no defence against *Denial-of-Service (DoS)* attacks, which are a common class of attack, and it also needs to get better even in some points of authentication security, as seen in the beginning of this section, with the reinforcement of the quality of the user's passwords and with restricting the number of login attempts in a single connection.

Chapter 5

The Implementation

This chapter is intended to dig a little deeper than the last chapter, and enter the implementation details. However, besides only detailing a little more the security requirements that were implemented in this work, this chapter will finally approach the architecture and implementation of the Security Interface, which was what made this work's solution flexible — which is a point that have been mentioned since the beginning of this paper.

5.1 Security Interface

Even though this work had indeed implemented security modules that satisfied the security requirements that were mentioned in the last chapter — authentication, authorization and transport layer security —, one of the focuses of this work lied in developing a **flexible** interface that could make it easy for anyone to implement and use, in AngraDB, other security modules that satisfy these very requirements.

The strategy that was adopted to build the interface was the same for both the authentication and authorization requirements: to use the Behaviour Design Pattern (which was explained in the section 2.2) and create two generic modules that, according to the Behaviour Design Pattern, are going to be the behaviour modules, and serve as *interface of creation* of new authentication or authorization modules, which will be the respective callback modules.

Important: The expression *interface of creation*, which will be repeated often, is key to the Security Interface logic. It comes from the usage of behaviour modules in AngraDB, which were thought to be exactly an interface that dictates how a new module should be implemented (created), in order be attached and used in AngraDB.

This strategy, as explained in the second chapter, is very common in Erlang, and even existed already in AngraDB. As well as Erlang exposes its Generic Server (*gen_server*) — which is a behaviour module — to be implemented by other callback modules, An-

graDB also followed this same strategy for the first time when the Generic Persistence (*gen_persistence*) module was exposed in AngraDB's API. This AngraDB's first behaviour module served as an *interface for the creation* of different kinds of persistence modules, such as the ones that we have nowadays: *adbtree_persistence*, *hanoidb_persistence* and *ets_persistence*, which are all callback modules of the *gen_persistence* behaviour module.

Back again to the Security Interface logic, the Generic Authentication and Generic Authorization modules — *gen_authentication* and *gen_authorization*, respectively — are going to be used by AngraDB **internally**. The respective callback modules are then going to have their functions — callbacks — called inside *gen_authentication* and *gen_authorization*. This way, the callback modules, which are implementations of specific authentication or authorization mechanisms, will be invisible to AngraDB internally, since it is only going to deal directly with the generic modules.

A good example of that would be a *login* operation: AngraDB would call the *login* function from *gen_authentication* module, and, inside this function, *gen_authentication* would call the *handle_login* function, which should be implemented by the authentication callback module, as specified in *gen_authentication*'s *behaviour_info* (the block of code inside a behaviour module that specifies the functions that should be implemented by the callback module and also their arities). To make things clearer, check Figure 5.1 and Figure 5.2.

```
-module(gen_authentication).

-export([behaviour_info/1]).

-export([start/2, login/4, logout/2, register/4]).

behaviour_info(callbacks) ->
  [{init, 2}, {handle_login, 3}, {handle_logout, 1}, {handle_register, 3}].
```

Figure 5.1: The top of the *gen_authentication* behaviour module. Note its *behaviour_info* and its exported functions.

Once the callback module is implemented, the only two things left to do are: configure the respective setup function inside the Server Supervisor (*server_sup*) — for example, if a new authentication callback module was added, you should configure the *setup_authentication* function, adding the new module as a possible option of authentication —, and put the name of the callback module and its configurations in AngraDB start-up options (inside the file "*adb_core.app*") in order to start using it. (Check the files "*adb_core.app*" and "*server_sup.erl*", inside the AngraDB repository, to get the idea a little better).

```
-module(adb_authentication).

-behaviour(gen_authentication).

% gen_authentication callbacks
-export([
  init/2,
  handle_login/3,
  handle_logout/1,
  handle_register/3
]).
```

Figure 5.2: The top of the *adb_authentication* callback module. Note its exported functions.

Summary of the steps needed to add a new security module (either authentication or authorization) to AngraDB and start using it:

- Implement the callback module for either Generic Authentication (*gen_authentication*) or Generic Authorization (*gen_authorization*) behaviour modules;
- Configure the respective setup function (*setup_authentication* or *setup_authorization*) inside the Server Supervisor module (*server_sup*), in order to add your module as a possible option of authentication or authorization mechanism;
- Set the name of your callback module and its configuration proplist inside the AngraDB start-up settings file ("*adb_core.app*") to start using the new authentication or authorization module.

5.2 Authentication

Most of the changes that were made in the development of the authentication architecture are focused in the ADB Server module (*adb_server*), which, as said in chapter 2, is the AngraDB module that manages the external communication and uses Transmission Control Protocol (TCP) to do so.

Each instance of the ADB Server, which will be taking care of a specific connection, has an internal state that is updated after handling each request. Until then, this state was being used to store information such as the socket of the current connection, the persistence module setup and the database that the user is currently connected.

This server state was then chosen to store the authentication information — both the authentication setup (which is a tuple with the authentication callback module that is being used and its settings) and also the authentication status of the session (a tuple with the authentication status itself, like *loggedIn* or *loggedOut*, and the user information, such as the username, in case he/she is already authenticated).

By having this authentication information available in the server status, it can now be used to restrict some actions of the user, and so was done.

When the server receives a message, it first parses the whole string to obtain the command and its arguments. After that, the commands used to be evaluated directly (within a big switch case that, based on the string of the command, executes a specific function), but that was the exact part that changed in the implementation of the authentication scheme: the old function that evaluated the commands right after the parsing (*evaluate_request*) was split in two new functions — *evaluate_authenticated_request* and *evaluate_not_authenticated_request*. Now, before the evaluation of the command, the authentication status of the connection is verified and, depending on the status, the function *evaluate_authenticated_request* or *evaluate_not_authenticated_request* will then be called.

Commands that do not depend on the authentication of the user (such as the *login* command) are evaluated inside the function *evaluate_not_authenticated_request*, and all the rest is evaluated in *evaluate_authenticated_request*. This way, if a user that is not logged in yet send a *create_db* command, for example, it will be evaluated inside the *evaluate_not_authenticated_request* function and, since it is a command that needs the user to be authenticated, the server will send an error message back to the user.

These two structural changes — the inclusion of the authentication setup and the authentication status in the server state, and the separation of the function *evaluate_request* into *evaluate_authenticated_request* and *evaluate_not_authenticated_request* — were the main changes that were made for the common authentication part inside the ADB Server.

Besides those two changes, the authentication commands (*login*, *logout* and *register*) were also added in the evaluation functions as possible commands. Remembering that, when these commands are evaluated, their respective functions are called **from the Generic Authentication behaviour module**, which, then executes the functions from the authentication callback module that is being used (and is specified in the server state, inside the authentication setup).

Both Challenge-Response Authentication and the Certificate-Based Authentication were implemented in callback modules of the Generic Authentication behaviour module, and, as callback modules, they needed to implement the functions that were defined by its behaviour module (described in its *behaviour_info*), which are the following callbacks:

- *init(AuthenticationSettings, PersistenceSetup)* — function that will be called by *gen_authentication* at the **initialization** of the application;
  - AuthenticationSettings: a proplist with the settings of the authentication module;

  - PersistenceSetup: a tuple with the in-use persistence module and its settings proplist;
  - Expected Return: the atom *ok* or an error expression.

- *handle_login(LoginArgs, PersistenceScheme, Socket)* — function that will be called by *gen_authentication* when a **login** command is executed;
  - LoginArgs: a tuple containing the login arguments (e.g. username and password);
  - PersistenceScheme: the in-use persistence module;
  - Socket: the socket of the current connection;
  - Expected Return: a tuple with the new authentication status (*loggedIn* or *loggedOut*) and the user's authentication information (which currently is only the username).

- *handle_logout(AuthenticationStatus)* — function that will be called by the module *gen_authentication* when a **logout** command is executed;
  - AuthenticationStatus: a tuple with the current authentication status itself (probably *loggedIn*) and the user's authentication information (which currently is only the username);
  - Expected Return: a tuple with the new authentication status (which will probably be *loggedOut*) and the user's authentication information (which will probably be *none*).

- *handle_register(RegisterArgs, PersistenceScheme, Socket)* — function that will be called by *gen_authentication* when a **register** command is executed;
  - RegisterArgs: a tuple containing the register arguments (e.g. username and password);
  - PersistenceScheme: the in-use persistence module;
  - Socket: the socket of the current connection;
  - Expected Return: a tuple with *ok* and the username or a tuple with *error* and an expression with the error cause.

5.2.1 Challenge-Response Authentication

This challenge-response authentication mechanism was implemented in a module called ADB Authentication (*adb_authentication*). As it is a callback module of the Generic

Authentication, this subsection will basically discuss how each of its functions were implemented.

init(AuthenticationSettings, PersistenceSetup)

This function just checks if the authentication database already exists. If it does not exist yet, a new one is created, using the persistence module that was set in AngraDB start-up configurations.

handle_login(LoginArgs, PersistenceScheme, Socket)

Since the keys to find the documents of the users inside the Authentication database are SHA-256 hash values of the usernames, that is the first thing this function will do. After the SHA-256 digest of the username is computed, this function uses it to retrieve the user's document from the Authentication database via the persistence module that was passed as parameter (PersistenceScheme).

In case the document is not found, it means that the user with this username has not been registered yet, so the authentication fails and the function returns the tuple *{loggedOut, invalid_password_or_username}*.

In case the document is found, it is time to verify the response to the challenge, that is, to check if the password is correct. The hash of the the user's salted password and the salt are then retrieved form the document.

After that, the password that has been typed by the user is passed to the PBKDF2 function along with the salt that was retrieved from the user authentication document. The PBKDF2 function will be executed using SHA-512, being iterated 150,000 times, and generating then a 512-bit digest, which will be compared to the one that was retrieved from the document.

If the just-calculated digest is different from the one in the document, it means that the password is not correct, so the function fails, and then returns the same tuple: *{loggedOut, invalid_password_or_username}*. This error is the same for both invalid username and invalid password in order to avoid giving sensitive information for a potential adversary.

In case the digest that was calculated moments ago matches with the digest that was stored in the user authentication document, the function returns the tuple *{loggedIn, Username}*, as a sign of success on the authentication.

Note: Since every document in AngraDB needs to be stored in JSON format, the Erlang library "jsone" was used to transform JSON into Erlang proplists and vice-versa.

handle_logout(AuthenticationStatus)

This function only returns the tuple *{loggedOut, none}*.

handle_register(RegisterArgs, PersistenceScheme, Socket)

This function looks a lot like *handle_login*. First, it calculates the SHA-256 hash of the username, in order to use it as the key of the user document that will be saved in the authentication database.

After that, a pseudo-random 22-character alphanumeric salt is generated. Having the password (which is inside the "RegisterArgs" parameter) and the salt in hands, the PBKDF2 function is applied with the same parameters as in *handle_login*: SHA-512 as hash function, 150,000 iterations, and a 512-bit digest as output.

With the hash value of the salted password and the salt itself, the document is ready to be created. Using the library *jsone*, the proplist with the digest of the salted password and the salt are converted to JSON format, and, using the persistence module that was set in the application start-up (and passed to this function via the "PersistenceScheme" parameter), the JSON document is saved in the authentication database, using the SHA-256 hash of the username as the document key.

5.2.2 Certificate-Based Authentication

The module that implemented this mechanism in AngraDB is called ADB Certificate Authentication (*adb_cert_authentication*). As certificate-based authentication counts on the SSL/TLS protocol to authenticate the user with his/her certificate, the only thing this module needs to do is store some minimum information of the user, which are going to be the username and the encoded certificate. Besides that, a list containing all the in-use usernames will be stowed in a separate document as well (this list will be used to prevent from repeating usernames while registering new users).

Disregarding the authentication scheme itself, the structure of how the documents are stored, retrieved and manipulated is almost the same than the *adb_authentication*, so the functions may look very similar too. Having all this said, we can now check the *adb_cert_authentication*'s implementations of the *gen_authentication* callback functions.

init(AuthenticationSettings, PersistenceSetup)

As well as in *adb_authentication* module, this function also checks if the authentication database exists. The difference is that, if it does not exist, besides creating a new one, it also creates a document for the list of usernames, as said before.

handle_login(LoginArgs, PersistenceScheme, Socket)

First, this function uses the Erlang SSL module to retrieve the client's certificate, using the Socket parameter. If it fails to retrieve the certificate — which is very unlikely, since the SSL/TLS handshake had to occur successfully in order to the connection to be established —, the login fails and the function returns the tuple *{loggedOut, SSLError}*.

If the certificate is retrieved, its SHA-256 digest is calculated, and using it as key, an attempt to retrieve the user's document is made in the authentication database. In case the document is not found, authentication fails and the tuple *{loggedOut, user_not_found}* is returned. Otherwise, the library *jsone* is used to decode the JSON document into a proplist, the username is retrieved, and the function returns the tuple *{loggedIn, Username}*, finishing the authentication.

handle_logout(AuthenticationStatus)

Only returns the tuple *{loggedOut, none}* as well.

handle_register(RegisterArgs, PersistenceScheme, Socket)

As well as in *handle_login*, the Erlang SSL module is used to try to retrieve the client's certificate. If it fails, registration fails and the tuple *{error, SSLError}* is returned.

After retrieving the client's certificate, its SHA-256 hash value is calculated, in order to be used as key of the user's authentication document as well. Just in case, the function verifies, using the persistence module given by the parameter "PersistenceScheme", if any document with this hash value already exists, that is, if this certificate has already been registered. If so, registration fails as well, and the tuple *{error, user_already_exists}* is returned.

Next, using the PersistenceScheme module, the document with the usernames is retrieved, and the username that was passed as an argument for the *register* command is searched in the list. If it already exists, the registration process fails and the tuple *{error, username_already_exists}* is returned.

After confirming that the given username is not in use, it is appended to beggining of the list of usernames, which is then encoded into JSON, and its document is updated in the database.

To finish the registration, the proplist with the username and the certificate is encoded into JSON, and it is then saved into the authentication database, using the PersistenceScheme module. If it is successfully saved, the function returns the tuple *{ok, Username}*; otherwise, the tuple *{error, ErrorType}* is returned.

5.3 Authorization

As well as the Authentication scheme, the Authorization architecture was mainly focused on the ADB Server module, which is the one that deals the external communication via TCP.

The first step was also very similar to the Authentication scheme: the authorization setup (the module and its settings) that will be used was also added to the state of the server, in order to be available to the functions of ADB Server when needed. (Normally, the callback modules are passed as parameter to the behaviour modules — that is, the Generic modules —, so that they can know the module from which they are going to execute the callbacks).

After the authentication scheme was implemented in ADB Server, right after the parse of a TCP message, the tokens of the command would be evaluated, and, depending on the authentication status of the user, the function *evaluate_authenticated_request* or *evaluate_not_authenticated_request* would be executed, receiving the parsed tokens with the command and its arguments as parameter.

It is in this point that the second step and major architectural change in ADB Server will happen in order to implant the new Authorization scheme.

Instead of immediately calling the function *evaluate_authenticated_request* or *evaluate_not_authenticated_request* after the verification of the authentication status of the user in the current connection, it will first be checked, **in case the user is already authenticated**, if the he/she has permission to perform the command that is going to be evaluated, using the function *request_permission* of the Generic Authorization module (*gen_authorization*). If the access is allowed, then the evaluation process of the command continues, otherwise, a message is sent back to the user, saying that the access to the desired action was denied. If the user was not authenticated yet, *evaluate_not_authenticated_request* is called normally, without any permission request.

Other than that, but still talking about the changes in ADB Server, the other exported functions of the Generic Authorization module, such as *grant_permission* (grants a permission to a specific user), *revoke_permission* (revokes the permission of a specific user) and *show_permission* (shows the permission of a specific user), were added to the list of evaluable commands inside the function *evaluate_authenticated_request*.

Now that the changes in ADB Server were discussed, it is possible to talk about what is necessary to develop an authorization callback module for AngraDB.

In order to create a new Authorization module, it will be necessary to implement the following callbacks of the Generic Authorization behaviour module (which are specified in its *behaviour_info*):

- *init(AuthorizationSettings, PersistenceSetup)* — function that will be called by *gen_authorization* at the **initialization** of the application;
  - AuthorizationSettings: a proplist with the settings of the authorization module;
  - PersistenceSetup: a tuple with the in-use persistence module and its settings proplist;
  - Expected Return: the atom *ok* or an error expression.
- *handle_request_permission(PersistenceScheme, Database, Username, Tokens)* — function that will be called by *gen_authorization* before the evaluation of a command, to see if the user has permission to execute a certain action or not;
  - PersistenceScheme: the in-use persistence module;
  - Database: the current connected database;
  - Username: the username of the user that is authenticated in the current connection;
  - Tokens: a tuple with strings of the command and its arguments (e.g. *{"create_db", "MyDatabase"}*);
  - Expected Return: a tuple with the result of the permission request (*granted* or *forbidden*) and an expression that will hold any extra information.
- *handle_grant_permission(PersistenceScheme, Database, GrantArgs)* — function that will be called by the module *gen_authorization* when a **grant_permission** command is executed;
  - PersistenceScheme: the in-use persistence module;
  - Database: the current connected database;
  - GrantArgs: a tuple with the arguments that were passed by the user along with the *grant_permission* command;
  - Expected Return: a tuple with *ok* and the permission that was granted, or *error* and an expression with the error cause.
- *handle_revoke_permission(PersistenceScheme, Database, RevokeArgs)* — function that will be called by *gen_authorization* when a **revoke_permission** command is executed;
  - PersistenceScheme: the in-use persistence module;
  - Database: the current connected database;

  - RevokeArgs: a tuple with the arguments that were passed by the user along with the *revoke_permission* command;
  - Expected Return: a tuple with *ok* and the permission that was revoked or a tuple with *error* and an expression with the error cause.

- *handle_show_permission(PersistenceScheme, Database, Username)* — function that will be called by *gen_authorization* when a **show_permission** command is executed;
  - PersistenceScheme: the in-use persistence module;
  - Database: the current connected database;
  - Username: the username of the user whose permission will be shown;
  - Expected Return: a tuple with *ok* and the user's permission on the specified database or a tuple with *error* and an expression with the error cause.

5.3.1 Discretionary Access Control (DAC) Implementation

This access control mechanism that was developed is inside a module called Simple Authorization (*simple_authorization*).

As it uses *Access Control Lists (ACLs)* as strategy to store the user's permissions, the Simple Authorization module will organize the permissions in documents that will be stowed in the Authorization database. Each of these documents will represent a specific database, and, inside each of them, there will be a list (proplist) composed by tuples of users and their permissions upon this specific database.

Even though the permissions are stored in these documents as integers, as explained in the previous chapter, the client, when using a command such as *grant_permission*, will refer to a permission in its abbreviated form: "owner permission" ≡ "o"; "read and write permission" ≡ "rw"; "read permission" ≡ "r"; "no permission" ≡ "n" or anything that is not one of the previously mentioned.

An important point about how this module treats the permissions is that, in order to know what permissions are required for each command, a constant list called *CommandsAndPermissions* was created, containing tuples with the name of the command and the required permission (as integers).

Another peculiarity about this authorization module that is important to mention is that, when a new database is created, its respective authorization document is only created when someone requests permission related to this very database. When done so, this user that requested the permission for the first time earns owner permission of this database.

Now it is time to talk about how this authorization module implemented the callbacks specified in the Generic Authorization behaviour module.

init(AuthorizationSettings, PersistenceSetup)

This function verifies if the authorization database has already been created. If it has not yet, a new one is created, and authorization documents are also created for sensitive databases — the authorization database and the authentication database. This is done to avoid problems with the peculiarity that was mentioned, such as, for example, someone trying to connect to one of these sensitive databases and earning owner permission upon them.

handle_request_permission(PersistenceScheme, Database, Username, Tokens)

This is probably the main function that the callback modules need to implement, since it is the function that indeed applies the access control.

As usual, the first thing to be done is calculate the SHA-256 hash value of the database name, which is then used as key to retrieve the authorization document of this database.

As stated before, if the databse's authorization document is not found, a new one is created, containing a list of permissions with one entry: the user that first requested permission to this database and its new owner permission.

If the authorization document is found, the JSON is decoded into a proplist. After that, the function checks in this list if the user has any permission upon this database (if none was found, it is considered that the user has no permission). Then, based on the token of the command that was intended to be executed (extracted from the Tokens parameter), its required permission is retrieved from the *CommandsAndPermissions* list, and, finally, the two permissions are compared (the user's permission and the command's required permission).

If the user's permission is greater or equal than the one required for the command, then the access is allowed and the function returns the tuple *{granted, User_permission_string}*; otherwise, the access is denied, and the function returns the tuple *{forbidden, ErrorMessage}*.

handle_grant_permission(PersistenceScheme, Database, GrantArgs)

First, the SHA-256 hash of the database name is computed. After that, the permission that was passed as argument, which is in the abbreviated format, is converted to integer — "o" turns into 3, "rw" into 2, "r" into 1 and "n" into 0.

With the hash value, the function retrieves the document of permissions using the module given by PersistenceScheme, decodes the JSON into a proplist, and tries to find the user in this list.

If the user and its permission is found in the proplist, its entry is replaced, having its permission updated. After that, the list is encoded into JSON again and the document is updated in the authorization database using the PersistenceScheme module.

However, if the user is not found in the proplist, a tuple with the username and the desired permission is appended to the beginning of the list, then it is encoded into JSON again, and the document is updated in the authorization database using the PersistenceScheme module.

After one of these two possibilities, the grant ends successfully, and the function returns the tuple *{ok, GrantedPermission}*.

handle_revoke_permission(PersistenceScheme, Database, RevokeArgs)

Again, the SHA-256 digest of the database name is calculated, and then used as key to retrieve the authorization document of the specific database, using the persistence module given by PersistenceScheme.

After retrieving the document, it is decoded into a proplist. The entry of this user is deleted from this proplist of permissions, then the list is encoded back into JSON, and the document is finally updated in the authorization database via the PersistenceScheme. After that, the revocation is succeeded, and the function returns the tuple *{ok, RevokedPermission}*.

If any error occurs in mean time, during, for example, the retrieval of the document, the revocation fails, and the function returns the tuple *{error, ErrorMessage}*.

handle_show_permission(PersistenceScheme, Database, Username)

The authorization document is retrieved via the PersistenceScheme module, using the SHA-256 hash value of the database name. After that, the JSON is decoded into a proplist. From this proplist, the user entry is retrieved, and then the function returns a tuple with an *ok* and the permission of this user — *{ok, UserPermission}*. It is important to mention that, if no entries were found for the specified user, the variable "UserPermission" of the tuple will hold the value "NoPermission".

If the authorization document could not be retrieved in the beginning of the function, it will return the tuple *{error, ErrorMessage}*.

5.4 Transport Layer Security

This section will be a little shorter than the first two, since, for this requirement, it was not developed an interface of creation that enables other people to implement new modules and attach them in AngraDB (which was the case for Authentication and Authorization). For the problem of security of the transport layer, all the efforts were taken to add the SSL/TLS protocol as a possible configuration in AngraDB.

This goal was achieved by using Erlang tools themselves, which provide a SSL module (*ssl*) that includes functions, for example, to listen to new connections, to accept a new connection, to send messages, to perform the SSL/TLS handshake and so on.

The changes started in the AngraDB start-up configurations. There, it is now possible to add a SSL entry to the settings proplist. In this entry, there will be the paths to the files that are needed to make the SSL protocol work, which are the server certificate, the file with its private key, and the file with its CAs' certificates.

Summarizing this first step: to set AngraDB to use SSL/TLS, it is necessary to add an "ssl" entry in the AngraDB start-up proplist (inside the file "*adb_core.app*"), containing the paths to the certificates and keys that are necessary to make the SSL/TLS protocol work, as it can be observed in Figure 5.3.

```
[
  {persistence, {{name, adbtree}, [{max_index_size, 50000000}]}},
  {authentication, {{name, adb_authentication}, []}},
  {authorization, {{name, simple_authorization}, []}},
  {ssl, [{certfile, "./test_certificates/server_cert.pem"}, {keyfile, "./test_certificates/server_key.pem"}, {cacertfile, "./test_certificates/ca_cert.pem"}]}
]
```

Figure 5.3: Example of AngraDB start-up settings, including SSL configurations..

These configurations will be used in the initialization of Server Supervisor (*server_sup*). Before, the Erlang TCP module (*gen_tcp*) was started at the initialization of AngraDB. Now, inside a function of the Server Supervisor called *setup_communication*, instead of immediately starting the TCP module, an attempt to retrieve the SSL configurations is made. If no configuration is found, then TCP module is initialized as it was before. Otherwise, the SSL module is started with the configurations that were given.

After that, the Server Supervisor, when instantiating a new ADB Server, will also include the communication module that will be used (either *gen_tcp* or the *ssl* module) to the list of parameters for the ADB Server instantiation. This communication module, as well as the other modules that were passed as parameter to the new instance of the ADB server, is now also stored in the ADB Server state.

Inside the ADB Server, when initializing, it used to immediately accept a new TCP connection (this *accept* function from the *gen_tcp* module is a blocking function that basically waits for a new connection and accepts it when an attempt of connection is

made). Now, depending on the communication module that is being used (*gen_tcp* or *ssl*), it will either accept a TCP connection directly, or accept a SSL connection **and** perform the SSL/TLS handshake.

The last change lies on how ADB Server used to send messages to the connected client. Before, the messages were all sent by using the *gen_tcp:send* function. Now, the server state will tell what communication module will be used to call the *send* function. For example, if SSL is set as the in-use communication module in the server state, then *ssl:send* will be used; otherwise, *gen_tcp:send* will be used.

Chapter 6

Conclusion

Developing the first security design for AngraDB was quite challenging, but it was possible to learn a lot from these very challenges. Security, cryptography, programming in Erlang and working in an open source project are just a few examples of the things that I have learned or improved with this work.

Regarding what was implemented, there are some things that need adjustments. Some messages and even API are still a little inflexible. For instance, although the *login* command only needs the username as argument when using Certificate-Based authentication, the server still sends a message saying that, in order to authenticate, the user should use the command "login [username] [password]". Another example is the API of the Generic Authorization behaviour module, which uses "permissions" in names of the functions, even though there some access control models that do not use "permissions", such as the Role-Based Access Control (RBAC).

Furthermore, creating a new module to be used in AngraDB could be even easier if it was not necessary to change any AngraDB existing code when doing so. Currently, in order to use a new module — either persistence or security module —, it is required to add this very module as an option on the functions *setup_persistence*, *setup_authentication* or *setup_authorization*, inside the Server Supervisor module, besides setting up this desired module in AngraDB start-up configurations. Instead, things will be much easier and less bug-prone if the only necessary step was changing the AngraDB start-up configurations (which are not even in the *source* folder of the repository).

Also, as stated previously in the section 4.4 (Evaluation Based On Attack Models), even though this work has been a breakthrough for AngraDB in terms of security, it still has **a lot** more to improve, both regarding what is already done (by enhancing the existing security modules and discovering new flaws and breaches on the current implementation, for example), and also in regard to adding new security modules (such as an Audit, an Encryption, or a Role-Based Authorization module) and new features to prevent attacks

that were not thought in this work, such as the mentioned *Denial-of-Service (DoS)* attack, for instance.

Besides that, security is a matter that is in constant change: vulnerabilities on the safest protocols are found; new stronger cryptographic algorithms are released; computational power is increased every year. Having this said, we can conclude that there are always new improvements to be done.

Note: All the code that has been developed in this work can be found in the AngraDB repository, on the *security* branch (`https://github.com/Angra-DB/core/tree/security`).

Bibliography

[1] E. McDonough, James: *Template Method Design Pattern*, chapter 19, pages 247–254. Apress, 2017, ISBN 978-1-4842-2837-1. https://www.researchgate.net/publication/318150037_Template_Method_Design_Pattern. viii, 6, 7

[2] Preneel, Bart: *Cryptographic hash functions*. European Transactions on Telecommunications, 5(4):431–448, 2010. viii, 13

[3] *Cryptographic hash function: A simple introduction*. https://steemit.com/steemstem/@leoumesh/cryptographic-hash-function-a-simple-introduction, visited on 2018-01-12. viii, 14

[4] Thion, Romuald: *Chapter xxxvii access control models*. Cyber Warfare and Cyber Terrorism, 2007. viii, 31, 32

[5] *Transmission control protocol specification*. https://tools.ietf.org/html/rfc793, visited on 2018-07-18. 4, 9

[6] *Erlang documentation*. https://www.erlang.org/docs, visited on 2018-11-10. 4

[7] Jonathan Katz, Yehuda Lindell: *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007, ISBN 1584885513. 12, 17, 19

[8] Stinson, Douglas R.: *Cryptography: Theory and Practice*. CRC PRESS, 2005, ISBN 15848850848. 12, 19

[9] Goldreich, Oded: *Foundations of Cryptography - Fragments of a Book*. Oded Goldreich, 1995. http://www.wisdom.weizmann.ac.il/~oded/frag.html. 12

[10] Bellare, Mihir and Phillip Rogaway: *Introduction to modern cryptography*. In *UCSD CSE 207 Course Notes*, page 207, 2005. 12

[11] Microsoft: *Hash indexes*. https://docs.microsoft.com/en-us/sql/database-engine/hash-indexes?view=sql-server-2014, visited on 2018-11-10. 12

[12] Menezes, Alfred J., Paul C. van Oorschot, and Scott A. Vanstone: *Handbook of Applied Cryptography*. CRC Press, 2001. http://www.cacr.math.uwaterloo.ca/hac/. 12, 13, 14, 17, 19

[13] Rogaway, Phillip and Thomas Shrimpton: *Cryptographic hash-function basics: Definitions, implications and separations for preimage resistance, second-preimage resistance, and collision resistance.* IACR Cryptology ePrint Archive, 2004:35, 2004. 12, 13

[14] *Comparison of cryptographic hash functions.* https://en.wikipedia.org/wiki/Comparison_of_cryptographic_hash_functions, visited on 2018-01-14. 14

[15] *Google kills sha-1 with successful collision attack.* https://www.infoworld.com/article/3173845/encryption/google-kills-sha-1-with-successful-collision-attack.html, visited on 2018-01-15. 15

[16] *Google just 'shattered' an old crypto algorithm – here's why that's big for web security.* https://www.forbes.com/sites/thomasbrewster/2017/02/23/google-sha-1-hack-why-it-matters/#1f0c98fb4c8c, visited on 2018-01-15. 15

[17] *'first ever' sha-1 hash collision calculated. all it took were five clever brains... and 6,610 years of processor time.* https://www.theregister.co.uk/2017/02/23/google_first_sha1_collision/, visited on 2018-01-15. 15

[18] Zulfany Erlisa Rasjid, Benfano Soewito, Gunawan Witjaksono Edi Abdurachman: *A review of collisions in cryptographic hash function used in digital forensic tools.* Procedia Computer Science, 116:381–392, 2017. 15

[19] *Length extension attack.* https://en.wikipedia.org/wiki/Length_extension_attack, visited on 2018-01-15. 15

[20] Security, Defuse: *Salted password hashing - doing it right.* https://crackstation.net/hashing-security.htm, visited on 2018-02-16. 16

[21] Tom St Denis, Simon Johnson: *Public Key Algorithms*, chapter 9, pages 379–407. Syngress, 2006, ISBN 978-1-59749-104-4. https://www.sciencedirect.com/science/article/pii/B9781597491044500121. 17, 19

[22] Cryptomathic: *What is non-repudiation?* https://www.cryptomathic.com/products/authentication-signing/digital-signatures-faqs/what-is-non-repudiation, visited on 2019-01-19. 18

[23] Simson Garfinkel, Alan Schwartz, Gene Spafford: *Cryptography Basics*, chapter 7. O'Reilly, 2003, ISBN 0-596-00323-4. https://docstore.mik.ua/orelly/other/puis3rd/0596003234_toc.html. 19

[24] IBM: *Public key algorithms.* https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.icha700/icha700_Public_key_algorithms.htm, visited on 2019-01-18. 19

[25] Zhu, Sencun: *Cse597b: Special topics in network and systems security: Public key cryptography.* http://www.cse.psu.edu/~sxz16/teach/cse597b/Intro_Crypto_3.pdf, visited on 2019-01-18. 19

[26] Jain, Raj: *Public key algorithms*. https://www.cse.wustl.edu/~jain/cse571-07/ftp/l_08pka.pdf, visited on 2019-01-18. 19

[27] Harn, L. and J. Ren: *Generalized digital certificate for user authentication and key establishment for secure communications*. IEEE Transactions on Wireless Communications, 10(7):2372–2379, July 2011, ISSN 1536-1276. 19

[28] IBM: *What is a digital certificate*. https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.15/gtps7/s7what.html, visited on 2019-01-20. 19

[29] Rouse, Margaret: *X.509 certificate*. https://searchsecurity.techtarget.com/definition/X509-certificate, visited on 2019-01-20. 19

[30] Hunt, R.: *Pki and digital certification infrastructure*. In *Proceedings. Ninth IEEE International Conference on Networks, ICON 2001.*, pages 234–239, Oct 2001. 20

[31] *Mongodb security documentation*. https://docs.mongodb.com/manual/security/, visited on 2018-05-06. 27

[32] *Couchdb security documentation*. http://docs.couchdb.org/en/2.2.0/intro/security.html, visited on 2018-05-06. 27

[33] *Discretionary access control*. http://www.cs.cornell.edu/courses/cs5430/2015sp/notes/dac.php, visited on 2019-01-17. 31

[34] Abhishek Majumder, Suyel Namasudra, Samir Nath: *Taxonomy and Classification of Access Control Models for Cloud Environments*, pages 23–53. Springer, London, July 2014, ISBN 978-1-4471-6452-4. 31, 32

[35] Rick Kuhn, David Ferraiolo: *An introduction to role-based access control*. https://spaf.cerias.purdue.edu/classes/CS526/role.html, visited on 2019-01-17. 32, 33

[36] Olenski, Julie: *Ssl vs. tls - what's the difference?* https://www.globalsign.com/en/blog/ssl-vs-tls-difference/, visited on 2019-01-18. 38

[37] Rhee, Man Young: *Transport Layer Security: SSLv3 and TLSv1*, chapter 9, pages 325–351. Wiley, 2013, ISBN 9781118512920. https://ieeexplore.ieee.org/document/8044594. 38, 39

[38] Chou, W.: *Inside ssl: the secure sockets layer protocol*. IT Professional, 4(4):47–52, July 2002, ISSN 1520-9202. 38

[39] IBM: *An overview of the ssl or tls handshake*. https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm, visited on 2019-01-18. 38

[40] Microsoft: *Description of the secure sockets layer (ssl) handshake*. https://support.microsoft.com/en-us/help/257591/description-of-the-secure-sockets-layer-ssl-handshake, visited on 2019-01-18. 38

[41] Kemmerer, Chris: *The ssl/tls handshake: an overview.* `https://www.ssl.com/article/ssl-tls-handshake-overview/`, visited on 2019-01-18. 38